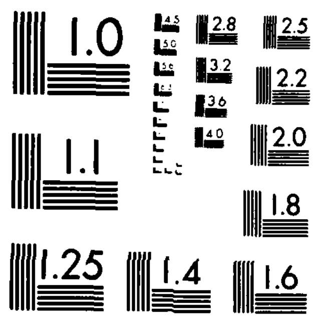
AD-A146 257 ADVANCED ADA (TRADEMARK)(U) SOFTECH INC WALTHAM MA 1/5
JUL 84 DAAB07-83-C-K514

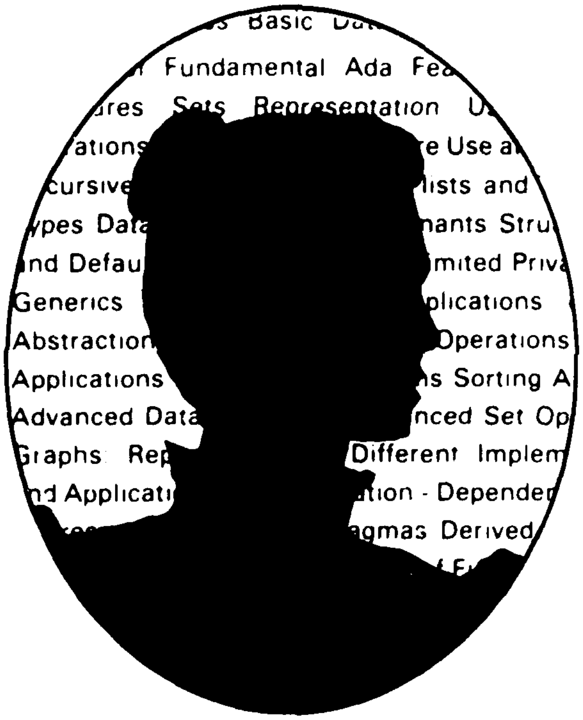
UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

JULY 1984

Advanced Ada



AD-A146 257

DTIC FILE COP

Contract DAAB07-83-C-K514

U.S. Army Communications-Electronics Command
(CECOM)
Center For Tactical Computer Systems
(CENTACS)

Prepared By:
SOFTECH, INC.
460 Totten Pond Road
Waltham, MA 02154

84 09 28 082

PREFACE

The Advanced Ada Workbook was developed under Contract Number DAAB07-83-C-K514 for the U.S. Army, Center for Tactical Computer Systems (CENTACS), Fort Monmouth, New Jersey.

This workbook is organized as a series of Exercises for the reader. Each exercise appears in the following format:

First the objective of the exercise is stated. It will prepare the students for the exercise by informing them what they will learn from the exercise.

A Tutorial section follows. This section provides the necessary background information for working the problem.

The Tutorial if followed by the Problem Statement. Here the student is given a situation in which he is to assume the role of "problem solver"; such as programmer, system designer, or maintainer.

Finally, there is a Discussion and Solution section in which possible solutions are discussed, their merits compared, and an ultimate complete solution chosen. The change in the approach from The Ada Primer Workbook is due to the complex nature of the problem statements. Due to the fact that the code for the solutions in this workbook is often several pages long, we find that the solution is easier to understand when the rationale for that solution is given with the solution. Also, because the solutions to these exercise are more that just producing code, some understanding of the design of the solution is needed for the reader to understand the solution. Thus our approach is to interleave some rationale and code, before giving a complete solution near the end of the section.

This workbook parallels and can be used in conjunction with L305, Advanced Ada Topics, of the U.S. Army Ada Training Curriculum. It is also intended as a follow-on for the Ada Primer Workbook. It assumes the reader is familiar with all the concepts covered in the Ada Primer. Exercise 1.1 is a review of the Ada features with which the reader of this workbook must be familiar. The reader, if not familiar with any topic in this Exercise, should review the Ada Primer before continuing with this workbook.

Letter on file

A-1

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER 1

REVIEW OF FUNDAMENTAL ADA FEATURES

EXERCISE 1.1

GENERAL REVIEW

Objective

This exercise provides concise review of some of the issues covered in the Ada Primer, so that the reader may be reacquainted with these basic features. This review also outlines the level of Ada that is assumed to have been learned by Advanced Ada Workbook readers. None of the material should seem new. Topics covered in the tutorial are: types, objects, control structures, subprograms, packages, separate compilation, visibility, and exceptions.

Tutorial

A complete Ada program consists of a main program in the form of a subprogram, with possible library units and subunits. Other program units, such as tasks (which will be addressed in the Real-Time Ada Workbook) or generic program units may also be attached. A subprogram has a declarative region, where entities are declared, followed by statements.

Types and objects are defined in the declarative region. A type declaration creates, in a sense, a blueprint from which an object can subsequently be formed. Because every type declaration introduces a distinct type, an object that is declared of one type may never be compared to an object of another type. Strong typing, as it is called, provides an extra check to ensure the correct processing of data.

A subtype is a subset of some type, and may be created from any type. A subtype declaration does not introduce a distinct type. It is an expression for a constrained (though the constraint is not mandatory) version of an existing type, where the constraint applies to the values of objects of that type, not the operations.

Predefined types and subtypes, such as Integer, conveniently allow the programmer to create an object without first defining its type. However, the range and specifications of these predefined types are implementation dependent, and their use can, therefore, decrease portability. Integer, Float, Boolean, Character, and String are all predefined types. The types universal integer and universal real are also predefined, but they cannot be referred to in a program. Natural (range 0 ... Integer'Last) and Positive (range 1 ... Integer'Last) are predefined subtypes.

The user may define specific types for particular kinds of data, scalar or composite. For numeric calculations, an integer type can be declared by specifying the range of values for objects of that type, as in

```
type Books_Count is range 0 .. 10_000;
```

Real numbers are represented using either floating point or fixed point forms. A floating point declaration contains the reserved word digits (the minimum number of significant decimal digits), and may optionally specify a range, for example,

```
type Angle_Type is digits 5;
type Cosine_Type is digits 6 range -1.0 .. 1.0;
```

There is a predefined type called Float, which has range and digits values that are defined by the implementation. So an object may be created,

```
Tangent : Float;
```

without defining a specific floating point type.

A fixed point type declaration contains the reserved word delta, (the relative precision) and a range constraint, as in

```
type Sine_Type is delta 0.025 range -1.0 .. 1.0;
```

There is only one predefined type for fixed point: Duration. It has a very specific usage, namely to measure time intervals (in seconds).

Enumeration types are declared by listing the names of the enumeration literals. An enumeration literal may be overloaded, as the literal Dry is in

```
type Terrain_Type is (Dry, Normal, Wet, Submerged);
type Martini_Type is (Dry, Extra_Dry, No_Vermouth);
```

When Dry is later refered to, it must be preceded by its type name,

```
Terrain_Type'(Dry)
```

if its type is not clear from the context.

The composite types, array and record, are used to group logically related parcels of data. Whereas each record component has a distinct name, array components are selected using index expressions, which can be computed at execution time. All of the components of an array are of the same type, whereas the components of a record may be of different types. The components of both array and record types may be of any constrained type (except when the component is a record with a discriminant and default value, a circumstance which will be addressed in Exercise 3.1). The index of an array must be of a discrete type (integer or enumeration).

Multi-dimensional array types, such as,

```
type Matrix_5x2_Type is array (Integer range 1 .. 5,
                               Integer range 1 .. 2) of Float;
```

specify two or more index subtypes, which need not be the same. Unconstrained array types,

```
type Message_Switch_Type is
          array (Integer range<>) of Boolean;
type Matrix_Type is array (Integer range<>,
                           Integer range<>,
                           Integer range<>) of Real;
```

have unbounded discrete indices. They can serve as a base type for constrained array subtypes, or in object declarations (as long as a constraint is specified). Given the type Matrix_Type declared above, one could declare the following subtypes and objects:

```
subtype Matrix_5x5x5_Type is
        Matrix_Type (1 .. 5, 1 .. 5, 1 .. 5);
Matrix_2x3x3 : Matrix_Type (1 .. 2, 1 .. 3, 1 .. 3);
```

An index constraint may not be applied to an array type that already has an index constraint. One could not now write,

```
subtype Matrix_2x2x2_Type is                          -- **ILLEGAL
        Matrix_5x5x5_Type (1 .. 2, 1 .. 2, 1 .. 2);
```

Array objects may also be declared using an anonymous array type. For example, the declaration,

```
Vector_System_1 : array (1 .. 5, 1 ..2) of Float;
```

creates an array that is similar to Vector_System_2, below,

```
type Matrix_5x2_Type is array (Integer range 1 .. 5,
                               Integer range 1 .. 2) of Float;

Vector_System_2 : Matrix_5x2_Type;
```

Vector_System_1 and Vector_System_2, although identical in index and component types, are of different types. It is also important to note that several objects declared of the same anonymous array type,

```
type Plane_Dimensions_Type is
        (Height, Wing_Span, Length, Capacity);
L1011_Dimensions, DC9_Dimensions :
        array (Dimensions_Type) of Real :=
                        (34.0, 96.3, 91.2, 198.0);
```

have distinct types and are, therefore, incompatible. The following statement is illegal:

```
L1011_Dimensions := DC9_Dimensions;          -- **ILLEGAL
```

although the component assignment below,

```
L1011_Dimensions(Capacity) := DC9_Dimensions(Capacity);
```

is allowed.

The use of a type name is preferable because it allows the programmer to create many arrays of the same type, and because the type may be used in a subprogram parameter list.

Record types are used to express a logical grouping of differently typed data. An individual record component declaration may contain a default value, but may not be constant (though an entire record, of course, may be). An anonymous array type may not be declared in a record declaration.

Arrays of records and records of arrays can be particularly useful structures. The following declarations,

```
type Boundary_Point_Type is                   -- point on boundary
    record
        X : Float := 0.0;                     -- default to (0.0, 0.0, 0.0)
        Y : Float := 0.0;
        Z : Float := 0.0;
    end record;

type Boundary_Type is                         -- map of boundary line
        array (Integer range 1 .. 500)
        of Boundary_Point_Type;

type Map_Type is                              -- array of all maps for
        array (Integer range 1 .. 100)        -- a certain area
        of Boundary_Type;

type Area_Map_Type is                         -- stores maps for a cer-
    record                                    -- tain area with area code
        Area_Code  : String (1 .. 5);
        Boundary   : Map_Type;
    end record;

type Map_File_Type is                         -- stores all maps
        array (Natural range <>)
        of Area_Map_Type;

Map_File : Map_File_Type (1 .. 500);
```

define a system for storing multiple contour maps of boundaries. Map_File_Type is an unconstrained array of a record type having two components: a string, and an array of an array of a record. The expression,

```
Map_File(269).Boundary(20)(10).Y
```

refers to the Y coordinate of the tenth point on the twentieth map for area number 269.

An if statement allows the conditional execution of certain statements. The example below,

```
if a = 0.0 then
    Compute_Linear_Equation;
elsif b**2 - 4.0 * a * c >= 0.0 then
    Compute_Real_Roots;
else
    Compute_Complex_Roots;
end if;
```

determines, by examining the coefficients, the roots of a quadratic equation. There may be several elsif parts in an if statement, and the final else part is optional.

Similarly, the case statement selects a sequence of statements (or just one) for execution according to the evaluation of an expression. Consider the case statement below:

```
procedure Computation is
    type Code_Type is (RE, RT, QF, AV, BX, OK, EY, KL, NC, JP, XX);
    Code : Code_Type := OK;
begin       -- Computation
        .
        .

    case Code is
        when RE =>
            RE_Evaluation;
        when RT .. AV =>
            Minor_Code_Evaluation;
        when BX | EY .. NC | XX =>
            Major_Code_Evaluation;
        when others =>
            No_Evaluation;
    end case;
        .
        .

end Computation;
```

All possible values for the discrete expression, in this case Code, must be handled in the choices, and the choices must be mutually exclusive. The choice others, which ensures that all possible values are represented, must occur, if it is used, as the last choice of the case statement.

Iterative control structures execute a specific sequence of statements zero or more times. The loop usually contains an iterative control scheme, either **while** or **for**. The **while** statement will repeatedly execute the specified sequence of statements as long as the condition of the iterative scheme is satisfied. The procedure in the following construct,

```
while Space_In_Queue
loop
    Add_Element_To_Queue;
end loop;
```

is repeatedly invoked until Space_In_Queue has the value False.

A **for** statement will execute the specified sequence of statements a certain number of times, that is, as long as the value of the loop parameter, which is automatically incremented after each successive execution of the sequence of statements, remains within the given discrete range. For example,

```
for I in Map_File'Range
loop
    if Map_File(I).Area_Code = "Tunis" then
        Add_Map (Map_File(I).Boundary);
    end if;
end loop;
```

The if statement here will execute until I exceeds the range of Map_File.

The type of the loop parameter in a **for** statement is determined by the range given. (The type of the range bounds must, therefore, be unambiguous.) The scope of the loop parameter extends only to the end of the loop, and it may not be assigned a value or be altered in any way by any of the loop statements. The reserved word **reverse** may be used to decrement, instead of increment, the loop parameter. However, the range bounds are not, in that case, written in descending order.

An **exit** statement can, when neither a **for** nor a **while** terminating condition is sufficient, be used to exit a loop. It may include a loop identifier, which is often used to specify which of several nested loops is to be exited. The **exit** may also be conditional, written, for example,

```
exit Search when Deactivation;
```

In this case, the loop marked by Search is exited if Deactivation is True.

The use of subprograms enables the programmer to separate a large program into parts, where each part serves a specific purpose. Such modularized software is likely to be easier to read and maintain. The two types of subprograms in Ada, functions and procedures, differ as follows: functions are called as components of expressions and return a result, whereas a procedure call is a statement in itself and does not evaluate to a result. Function and procedure specifications and bodies are declared in the declarative region of a block or of another subprogram, or in a package. A function or procedure is nested inside another subprogram when it is only useful to its parent.

The function specification includes a list of the *formal parameters* (if there are parameters), their respective types, and the type of the returned value. The procedure specification lists the formal parameters (again, they are optional), their respective types, their modes (**in**, **out**, or **in out**), and possible default values (for parameters of **in** mode only). The two subprogram declarations below,

```
function Lowest_Common_Denominator (X : Integer;
                                    Y : Integer := 0)
                                              return Integer;
type Time_Type is array (1 .. 50) of Real;
procedure Sort_Time (Time : in out Time_Type);
```

are separate from their bodies. The function actually returns a value of type Integer; the procedure makes changes to the array, which are then accessible outside of the procedure. The formal parameters of both functions and procedures may be of an unconstrained array type, in which case the bounds of the formal parameters are taken from those of the actual parameter (those specified in the subprogram call). In all cases, the formal parameters must be of named types.

The types of the formal parameters and actual parameters must always match. The actual parameters may be written in either named notation, positional notation, both named and positional (with regard to certain restrictions) or, if there are default values, some or all of the actual parameters can be omitted.

A function designator may be an operator written as a character string, as in the function,

```
type Vector_Type is array (1 .. 2) of Float;
function "+" (A, B : Vector_Type) return Vector_Type is
    Result : Vector_Type;
begin    -- "+"
    Result(1) := A(1) + B(1);
    Result(2) := A(2) + B(2);
    return Result;
end "+";
```

in which case the + symbol is said to be overloaded. The above function can be called by writing,

```
Z := X + Y
```

where X, Y, and Z are of Vector_Type.

Assignment to global variables (that is, variables not declared inside the procedure or function) is allowed in both functions and procedures, but it is not generally good practice. If a variable is to be changed, it ought to be passed as a parameter.

A package groups together related declarations and subprograms. The specification gives the interface to the procedures that use the package, and the body contains the specific contents of the package. Consider the following declaration:

```
package Vector_Arithmetic is
    type Real is digits 5 range -1000.0 .. 1000.0;
    type Vector_Type is array (1 .. 2) of Real;
    Unit_Vector : constant Vector_Type := (1.0, 1.0);
    function "+" (X, Y : Vector_Type) return Vector_Type;
    function "*" (X, Y : Vector_Type) return Vector_Type;
end Vector_Arithmetic;

package body Vector_Arithmetic is
    function "+" (X, Y : Vector_Type) return Vector_Type is
        ...
    end "+";
    function "*" (X, Y : Vector_Type) return Vector_Type is
        ...
    end "*";
end Vector_Arithmetic;
```

The specification of this package contains type and object declarations, and subprogram specifications. The package body contains the subprogram bodies; they may not be in the specification. Indeed, if there is a subprogram specification in the package specification, then the corresponding subprogram body must be in the package body. A subprogram whose specification is not in the package specification may appear in the package body, but the subprogram is then internal to the package. Although it is not the case in Vector_Arithmetic, a package body may also have statements, which are separated from the declarative part of the body by **begin**. The package itself is declared in the declarative region of a procedure or another package, unless it is a library or secondary unit.

Separate compilation provides two methods for writing programs: top-down and bottom-up. Top-down is used primarily to decompose large program units, where it is preferable to write the upper units first, with the subunits needed for certain processes "stubbed out" and written later. For example, if a unit called Grade_Calculations used a procedure Calculate_Median, one could write,

```
type Switch_Type is array (1 .. 25) of Boolean;
procedure Calculate_Median (X : in  Switch_Type;
                            Med : out Integer) is separate;
```

meaning the body of Calculate_Median is in a separate subunit, which, incidently, would commence,

```
separate (Grade_Calculations)
procedure Calculate_Median (X   : in  Switch_Type;
                            Med : out Integer) is
        .
        .

end Calculate_Median;
```

These secondary units may be subprogram bodies, package bodies, or subunits.

The bottom-up method is used, for example, to write system libraries that are used by many programs. Typically, these units are written before the procedures that use them. A library unit may be a package declaration, a subprogram declaration, or a subprogram body. They are accessed by using the **with** and **use** clauses. (A secondary unit may not appear in a **with** or **use** clause.) In general, compilation units must be compiled after the units upon which they depend. Thus, Grade_Calculations must be compiled before Calculate_Median.

The scope of a declaration is the region of program text that is potentially influenced by that declaration. The immediate scope of a declaration extends from the declaration to the end of the immediately enclosing region, which is called the parent of the declaration. Declarations that have an extended scope (which reaches to the end of the full scope of the parent) are: declarations in the visible part of the package, subprogram parameters, and record components.

The related concept of visibility determines where, within its scope, an identifier will be recognized as a reference to the entity associated with it in its declaration. Direct visibility is the ability to use the identifier alone to refer to the declared entity. A declaration is directly visible anywhere in its immediate scope, except where hidden by a different declaration. Visibility by selection occurs when the identifier must be accompanied by some qualifying prefix, such as,

```
Text_IO.Get
```

or must appear in a certain context to be recognized as an occurrence of the declared entity. A **use** clause used in conjunction with a **with** clause, as in

```
with Text_IO; use Text_IO;
procedure Report is
...
end Report;
```

makes the declarations in the specified package directly visible, eliminating the need for the qualifying prefix.

Exceptions are errors that arise during program execution. When the error occurs, the appropriate exception is raised and normal program execution is abandoned. There are several predefined exceptions: Constraint_Error, raised when a range, index, or discriminant constraint is violated; Numeric_Error, raised when a numeric operation cannot deliver an accurate numeric result; Storage_Error, raised when available storage is insufficient; Program_Error, raised when a function is exited by some means other than a return statement, or when a unit is activated whose body is not yet elaborated (for the purpose of this tutorial, written and compiled); and Tasking_Error, raised when an exception occurs during inter-task communication (to be covered in the Real-Time Ada workbook). There are also predefined exceptions for Text_IO, which will be explained in the next chapter.

Exception handlers specify some action that is to be performed in response to the raising of an exception. They appear just before the end of the program unit. Package bodies, subprogram bodies, task bodies, and block statements have exception handlers as an optional part of their structure. The example of an exception handler, below,

```
procedure Verify_Data is
        .
        .
begin           -- Verify_Data
        .
        .
exception
    when Data_Error =>
        Put_Line ("Incorrect Data");
    when others =>
        Put_Line ("Some error -- Program incomplete");
end Verify_Data;
```

prints a certain message if Data_Error is raised, and another message if any other exception is raised.

Exceptions may be user-defined, as in,

```
Divide_By_Zero : exception;
```

and may be explicitly raised, for example,

```
raise Divide_By_Zero;
```

Language defined exceptions can also be explicitly raised, but it is generally not good programming practice.

Problem

Write a package (specification and body) that is to be used by a program that converts quantities of foreign measures into the English system (inches, feet, yards, and miles). The program will accept the name of a foreign unit of measure and a real quantity of that unit. In this exercise, write only the package containing the necessary type, object, and subprogram definitions. The package should declare a table for the following values (it is not necessary to store the country):

Name of Unit	Country	Equivalent
Legua	Uraguay	3.200 mi.
Peninkulma	Finland	6.210 mi.
Ri	Japan	2.440 mi.
Chang	Mongolia	3.500 yds.
Cho	Japan	119.300 yds.
Kung chang	Taiwan	10.936 yds.
Rute	West Germany	4.120 yds.
Vara	Spain	2.740 ft.
Cubito	Somalia	1.833 ft.
Dhira	Syria	29.500 ins.
Dira	Saudi Arabia	17.300 ins.
Gazi jerib	Afghanistan	29.000 ins.
Hand	United Kingdom	4.000 ins.
Meter	France	39.370 ins.

There should be a function that converts, using the values in the table, a foreign measurement into English units. There should also be a function that adjusts the English units so that there are no decimal parts in the values for miles, yards, or feet (1.5 feet => 1.0 foot and 6.0 inches), and there are no excessive quantities in the values for inches, feet, or yards (93.1 inches => 2.0 yards, 1.0 foot, and 9.1 inches).

Solution and Discussion

The package described in the problem will contain the type and subprogram declarations needed for converting foreign units to English. First, let us consider the types for the objects that store English units and foreign units.

A measurement in English units will have four parts, inches, feet, yards, and miles. Note that these parts must be grouped as a single object. Even though the value of every component is real, an array type is not appropriate because each component has a different range. A record type should be used.

The component type represents the quantity of each unit, which is a real number. The predefined type Float could be used, but numbers of units cannot be negative, a fact that recommends the declaration of a user-defined type which allows only positive real values, thereby preventing the accidental assignment of erroneous negative values (this is called, as you remember, type checking). Defining one's own floating point type is preferable to using Float only in that the exact specifications of the type are known, which is useful when the code is used on a different machine. So, we can write,

```
type Positive_Real is digits 5 range 0.0 .. 5000;
```

to be used for quantities of a measurement. The record type for measurements in English units then appears,

```
type Measurement_Type is

      record

          Inches : Positive_Real range 0.0 ..   12.0;
          Feet   : Positive_Real range 0.0 ..    3.0;
          Yards  : Positive_Real range 0.0 .. 1760.0;
          Miles  : Positive_Real;

      end record;
```

The names for English measurements should be defined. An enumeration type consisting of the names of each unit, such as,

```
type English_Units_Type is (Inches, Feet, Yards, Miles);
```

is a clear representation, a notable advantage for the maintainer.

Foreign measurements, on the other hand, have three parts that are of different types. They are the name of the foreign unit, the English unit used in the conversion factor, and the real conversion factor. Measurements in foreign units should therefore be stored in an object of a defined record type that has three components corresponding to those three parts.

The names of the foreign units could also be represented by an enumeration type as in the declaration for the names of the English units. However, if an enumeration type were used, the list of foreign units available to the user could not easily be expanded. A string type, on the other hand, would not inhibit the expansion of the list.

Although a string could be declared inside the record type for the foreign measurement, it would be better declared separately because it is likely to be used in the subprograms that are defined in the package (as a parameter), or in the main program itself. Since String is a predefined type (an unconstrained array of characters), a user-defined (necessarily constrained) version of the type must be a subtype. So, we declare a subtype for the name of a foreign unit:

```
subtype Foreign_Unit_Name_Type is String (1 .. 10);
```

The upper range of 10 is chosen because it is the length of the longest foreign unit name listed in the problem.

The function that performs the actual conversion accepts two parameters, the name of a foreign unit and the quantity of that unit. It returns the equivalent English measurement, a Measurement_Type. So we can write the function specification:

```
function Conversion (Foreign_Unit_Name : Foreign_Unit_Name_Type;
                     Number_of_Units   : Positive_Real)
                     return Measurement_Type;
```

The package specification can now be written with the entities that have been defined:

```
package English_Measurements is

   type Positive_Real is digits 5 range 0.0 .. 5000.0;

   type English_Units_Type is                          -- English units.
         (Inches, Feet, Yards, Miles);

    type Measurement_Type is                           -- Measurements in
         record                                        -- English units.
            Inches : Positive_Real range 0.0 ..   12.0;
            Feet   : Positive_Real range 0.0 ..    3.0;
            Yards  : Positive_Real range 0.0 .. 1760.0;
            Miles  : Positive_Real;
        end record;

    Measure_Not_Found : exception;                     -- Raised when measure
                                                       -- not found in table.

    subtype Foreign_Unit_Name_Type is String (1 .. 10);

    function Conversion                                -- Converts a number of
      (Foreign_Unit_Name : Foreign_Unit_Name_Type;     -- foreign units to
       Number_of_Units   : Positive_Real)              -- English units.
       return Measurement_Type;

end English_Measurements;
```

The second element of the record type for foreign measurements, the English unit associated with the conversion factor (e.g. "Inches" in Dhira = 29.5 inches), is of the type already defined English_Units_Type. And the third element of the record type, the conversion factor itself, is a real type, in fact, necessarily positive. Therefore, it can be of type Positive_Real, as are quantities of units. Thus, the record type for foreign measurements is as follows,

```
type Foreign_Unit_Type is
    record
        Unit_Name         : Foreign_Unit_Name_Type;
        Conversion_Unit   : English_Units_Type;
        Conversion_Factor : Positive_Real;
    end record;
```

Given a record for each foreign unit, the table that stores the conversion information may conveniently be declared an array of those records. The index subtype should be of the predefined type Natural because, in this case, a negative index is meaningless. And, it ought to be unconstrained to allow the expansion of the table. So, we write,

```
type Conversion_Table_Type is array (Natural range <>) of Foreign_Unit_Type;
```

The table itself, that is, the object, must be constrained. It should be constant and initialized to an array aggregate of the values given in the problem, as follows,

```
Conversion_Table : constant Conversion_Table_Type (1 .. 14) :=
          (("Legua     ", Miles,  3.2),
           ("Peninkulma", Miles,  5.21),
           ("Ri        ", Miles,  2.44),
           ("Chang     ", Yards,  3.5),
           ("Cho       ", Yards,  119.3),
           ("Kung chang", Yards,  10.936),
           ("Rute      ", Yards,  4.12),
           ("Vara      ", Feet,   2.74),
           ("Cubito    ", Feet,   1.833),
           ("Dhira     ", Inches, 29.5),
           ("Dira      ", Inches, 17.3),
           ("Gazi jerib", Inches, 29.0),
           ("Hand      ", Inches, 4.0),
           ("Meter     ", Inches, 39.37));
```

The function Conversion should do the following: look up the conversion factor and associated English unit for Foreign_Unit_Name, multiply Number_of_Units by the conversion factor, store that value in an object of Measurement_Type, specifically in the part associated with the conversion factor, and then "cleanup" that object as described in the problem. Some of these maneuvers can be contracted out to separate subprograms. A second subprogram can look up Foreign_Unit_Name and return the conversion information for it, and another subprogram can adjust the values in the resulting object so that it is more readable. The main function then becomes quite simple:

```
function Conversion (Foreign_Unit_Name : Foreign_Unit_Name_Type;
                     Number_of_Units   : Positive_Real)
                     return Measurement_Type is

    Conversion_Factor   : Positive_Real;
    Conversion_Unit     : English_Units_Type;

begin      -- Conversion

    Look_Up_Conversion_Values
        (Foreign_Unit_Name, Conversion_Factor, Conversion_Unit);

    return Adjusted_Measurements
        (Number_of_Units * Conversion_Factor, Conversion_Unit);
end Conversion;
```

The two objects declared, Conversion_Factor and Conversion_Unit, store, remarkably, the conversion factor and unit that are returned by Look_Up_Conversion_Values. They are two of the three actual parameters listed in the procedure call.

Look_Up_Conversion_Values accepts (the in parameter) the name of a foreign unit, presumably listed in the table, and returns the corresponding factor and unit of conversion (the out parameters), both found in the conversion table. It must be a procedure, as opposed to a function, because it must return two values. Its specification will appear,

```
procedure Look_Up_Conversion_Values
        (Foreign_Unit_Name : in  Foreign_Unit_Name_Type;
         Conversion_Factor : out Positive_Real;
         Conversion_Unit   : out English_Units_Type);
```

The body of the procedure should contain a loop which searches the table for the requested unit name. It should give as its iterative scheme a for clause, specifying that the loop parameter should step through the range of Conversion_Table (which is obviously better than giving a range of 14, because the size of the table may well change). When Foreign_Unit_Name is found, the appropriate values are assigned to Conversion_Factor and Conversion_Unit. If Foreign_Unit_Name is not found in the table an exception is raised. The procedure is as follows:

```
procedure Look_Up_Conversion_Values
        (Foreign_Unit_Name : in Foreign_Unit_Name_Type;
         Conversion_Factor : out Positive_Real;
         Conversion_Unit   : out English_Units_Type) is

begin      -- Look_Up_Conversion_Values

    Conversion_Factor := 0.0;
    for I in Conversion_Table'Range
    loop
       if Conversion_Table(I).Unit_Name = Foreign_Unit_Name then
            Conversion_Unit   := Conversion_Table(I).Conversion_Unit;
            Conversion_Factor := Conversion_Table(I).Conversion_Factor;
       end if;
    end loop;

    if Conversion_Factor = 0.0 then
        raise Measure_Not_Found;
    end if;

end Look_Up_Conversion_Values;
```

Adjusted_Measurements accepts two parameters, a real quantity and a type of English unit. It returns a record of Measurement_Type with properly adjusted values. The specification for Adjusted_Measurements is written:

```
function Adjusted_Measurements
                 (Quantity : Positive_Real;
                  Unit     : English_Units_Type)
                  return Measurement_Type;
```

The body of Adjusted_Measurements can be divided into two sections. First, it checks each part of a Measurement_Type to see if there is a decimal part, which, if found, is converted to a number of lower units and added to the value currently in that lower entry. In other words, the decimal part of Miles is converted to some number of yards and added to the current value in Yards. That number of yards that is derived from the decimal part of Miles may well have a decimal part itself, which necessitates going down the array (Miles => Inches), rather than up.

To find the decimal part of the number with which the function is called, the whole part is subtracted from it. The whole part is determined by converting the number to Integer, which rounds, so it is necessary to subtract from it 0.5, to ensure that, in effect, the Integer conversion truncates. The whole part must then be converted back to Positive_Real, so that there is not a type mismatch when the subtraction is performed.

Since it is repeatedly necessary to reference the decimal part of the Positive_Real array element, and since it is not a simple step, a separate function, Decimal_Part, should be declared. It is written as follows,

```
function Decimal_Part (X : Positive_Real) return Positive_Real is
begin     -- Decimal_Part
    return X - Positive_Real (Integer (X - 0.5));
end Decimal_Part;
```

Because this function is only used by Adjusted_Measurements, it can be declared inside the procedure (i.e. it is internal), and its scope then extends only from its declaration to the end of the Adjusted_Measurements.

It is also necessary to define an intermediate type for the calculated results because the values of Miles, Yards, Feet, and Inches may not always satisfy the range constraints of the respective components of a record of Measurement_Type. The intermediate type has an index that corresponds to English_Units_Type, and components of type Positive_Real.

```
type Intermediate_Type is
    array (English_Units_Type) of Positive_Real;

Msmnt:  Intermediate_Type;
```

The first part of Adjusted_Measurements appears:

```
begin      -- Adjusted_Measurements

    Msmnt (Unit) := Quantity;
    if Decimal_Part (Msmnt (Miles)) > 0.0 then     -- Check for decimal
        Msmnt (Yards) := Msmnt (Yards) +           -- part in Miles value.
          Decimal_Part (Msmnt (Miles)) * 1760.0;   -- Convert and add to
    end if;                                        -- Yards.

    if Decimal_Part (Msmnt (Yards)) > 0.0 then     -- Same for Yards to
        Msmnt (Feet) := Msmnt (Feet) +             -- Feet, etc.
                Decimal_Part (Msmnt (Yards)) * 3.0;
    end if;

    if Decimal_Part (Msmnt (Feet)) > 0.0 then
        Msmnt (Inches) := Msmnt (Inches) +
                Decimal_Part (Msmnt (Feet)) * 12.0;
    end if;
```

The second part of Adjusted_Measurements converts excess values in Inches, Feet, and Yards, to Feet, Yards, and Miles, respectively, and adds appropriately to those values. If the value in Inches is greater or equal to 12.0, the number of times that 12 can be divided into Inches is added to Feet, and Inches is reduced by that many feet multiplied by 12. This method is repeated for Feet and Yards, with the difference that the decimal part in Inches must be preserved, whereas it may be assumed at this point, that all other units are whole. The second part of Adjusted_Measurements is:

```
if Msmnt (Inches) >= 12.0 then
    Msmnt (Feet) := Msmnt (Feet) +
        Positive_Real (Integer (Msmnt (Inches) - 0.5) / 12);
    Msmnt (Inches) := Msmnt (Inches) -
        Positive_Real (Integer (Msmnt (Inches) - 0.5) / 12) * 12.0;
end if;

if Msmnt (Feet) >= 3.0 then
    Msmnt (Yards) := Msmnt (Yards) +
            Positive_Real (Integer (Msmnt (Feet) - 0.5) / 3);
    Msmnt (Feet) :=
            Positive_Real (Integer (Msmnt (Feet) - 0.5) mod 3);
end if;

if Msmnt (Yards) >= 1760.0 then
    Msmnt (Miles) := Msmnt (Miles) +
            Positive_Real (Integer (Msmnt (Yards) - 0.5) / 1760);
    Msmnt (Yards) :=
            Positive_Real (Integer (Msmnt (Yards) - 0.5) mod 1760);
end if;

Result.Inches := Msmnt (Inches);
Result.Feet   := Msmnt (Feet);
Result.Yards  := Msmnt (Yards);
Result.Miles  := Msmnt (Miles);

return Result;
```

When Adjusted_Measurements is called only one component of the intermediate record will be assigned a value. Adjusted_Measurements could be simplified by allowing only for that situation. But written as it is, the function can be used if features are added to the program (such as addition and multiplication of English measurements), in which several or all of the indices of a Measurement_Type object may contain values. Because the function is in a package, it is a particularly good idea to make it flexible, because the package may be "with"ed for use in that many more programs.

Of the types, objects, and subprograms declared in the package, only Positive_Real, English_Units_Type, Measurement_Type, Foreign_Unit_Name_Type, and the function Conversion should appear in the package specification because only those entities need be visible. The other declarations, along with the body of Conversion (which may not appear in the package specification) appear in the package body. The reason that a subprogram body may not be in the specification of a package is that the exact method of the subprogram can then be concealed from the user, which prevents the possible abuse of that knowledge. The user does not know exactly how Conversion works, and therefore need not even be aware that the dependent procedures Look_Up_Conversion_Values or Adjusted_Measurements exist at all.

In the body of English_Measurements, Look_Up_Conversion_Values and Adjusted_Measurements must be declared before Conversion because they are called by Conversion. If they have not already been declared, an error will be cited when Conversion is compiled, because the compiler does not yet know what the two subprograms are. Keep in mind, however, that the order of the declarations in the package body would be unimportant if they were declared in the package specification. In other words, if we declare Look_Up_Conversion_Values and Adjusted_Measurements in the package specification, their bodies could be listed in the package body in any order whatsoever.

The code for the solution follows:

```
package English_Measurements is

    type Positive_Real is digits 5 range 0.0 .. 5000.0;

    type English_Units_Type is                      -- English units.
            (Inches, Feet, Yards, Miles);

    type Measurement_Type is
        record
            Inches : Positive_Real range 0.0 ..   12.0;
            Feet   : Positive_Real range 0.0 ..    3.0;
            Yards  : Positive_Real range 0.0 .. 1760.0;
            Miles  : Positive_Real;
        end record;

    Measure_Not_Found : exception;                  -- Raised when measure
                                                    -- not found in table.
    subtype Foreign_Unit_Name_Type is String (1 .. 10);

    function Conversion
       (Foreign_Unit_Name : Foreign_Unit_Name_Type;
        Number_of_Units   : Positive_Real)
                   return Measurement_Type;

end English_Measurements;
```

```
package body English_Measurements is

    type Foreign_Unit_Type is                          -- A unit name and its
        record                                         -- conversion factor.
            Unit_Name         : Foreign_Unit_Name_Type;
            Conversion_Unit   : English_Units_Type;
            Conversion_Factor : Positive_Real;
        end record;

    type Conversion_Table_Type is                      -- Table of all foreign
            array (Natural range <>)                   -- units and their
            of Foreign_Unit_Type;                      -- conversion factors.

    Conversion_Table : constant Conversion_Table_Type (1 .. 14) :=

                (("Legua     ", Miles,  3.2),
                 ("Peninkulma", Miles,  6.21),
                 ("Ri        ", Miles,  2.44),
                 ("Chang     ", Yards,  3.5),
                 ("Cho       ", Yards,  119.3),
                 ("Kung chang", Yards,  10.936),
                 ("Rute      ", Yards,  4.12),
                 ("Vara      ", Feet,   2.74),
                 ("Cubito    ", Feet,   1.833),
                 ("Dhira     ", Inches, 29.5),
                 ("Dira      ", Inches, 17.3),
                 ("Gazi jerib", Inches, 29.0),
                 ("Hand      ", Inches, 4.0),
                 ("Meter     ", Inches, 39.37));


-- Look_Up_Conversion_Values searches the table for the foreign unit name that
-- is passed to it, and returns the corresponding conversion factor and its
-- English unit.

    procedure Look_Up_Conversion_Values
            (Foreign_Unit_Name : in Foreign_Unit_Name_Type;
             Conversion_Unit   : out English_Units_Type;
             Conversion_Factor : out Positive_Real) is
```

```
begin       -- Look_Up_Conversion

    Conversion_Factor := 0.0;
    for I in Conversion_Table'Range
    loop
        if Conversion_Table(I).Unit_Name = Foreign_Unit_Name then
            Conversion_Unit   := Conversion_Table(I).Conversion_Unit;
            Conversion_Factor := Conversion_Table(I).Conversion_Factor;
        end if;
    end loop;

    if Conversion_Factor = 0.0 then
        raise Measure_Not_Found;
    end if;

end Look_Up_Conversion_Values;
```

```
-- Adjusted_Measurements adjust the values on an English measurement.  It first
-- converts the decimal parts of Miles, Yards, and Feet, to Yards, Feet, and
-- Inches, respectively, and adds to those values, (1.5 Yards =  1.0 Yards,
-- 1.0 Feet, and 6.0 Inches).

        function Adjusted_Measurements
                (Quantity : Positive_Real;
                 Unit     : English_Units_Type)
                 return Measurement_Type is

            type Intermediate_Type is
                array (English_Units_Type) of Positive_Real;

            Msmnt  : Intermediate_Type;
            Result : Measurement_Type;

            function Decimal_Part (X : Positive_Real) return Positive_Real is
            begin      -- Decimal_Part
                return X - Positive_Real (Integer (X - 0.5));
            end Decimal_Part;

        begin      -- Adjusted_Measurements

            Msmnt (Unit) := Quantity;

            if Decimal_Part (Msmnt (Miles)) > 0.0 then      -- Check for decimal
                Msmnt (Yards) := Msmnt (Yards) +            -- part in Miles value.
                  Decimal_Part (Msmnt (Miles) * 1760.0;     -- Convert and add to
            end if;                                         -- Yards.

            if Decimal_Part (Msmnt (Yards)) > 0.0 then     -- Same for Yards to
                Msmnt (Feet) := Msmnt (Feet) +              -- Feet, etc.
                        Decimal_Part (Msmnt (Yards) * 3.0;
            end if;

            if Decimal_Part (Msmnt (Feet)) > 0.0 then
                Msmnt (Inches) := Msmnt (Inches) +
                        Decimal_Part (Msmnt (Feet)) * 12.0;
            end if;
```

```
-- Next, excess values are appropriately distributed.  If Inches  = 12, or
-- Feet  = 3, or Yards  = 1760, convert extra to Feet, Yards, and Miles,
-- respectively, and add to those values, (e.g. 40.5 Inches =  1.0 Yards,
-- 4.5 Inches).

            if Msmnt (Inches) >= 12.0 then
                Msmnt (Feet) := Msmnt (Feet) +
                    Positive_Real (Integer (Msmnt (Inches) - 0.5) / 12);
                Msmnt (Inches) := Msmnt (Inches) -
                    Positive_Real (Integer (Msmnt (Inches) - 0.5) / 12) * 12.0;
            end if;
            if Msmnt (Feet) >= 3.0 then
                Msmnt (Yards) := Msmnt (Yards) +
                        Positive_Real (Integer (Msmnt (Feet) - 0.5) / 3);
                Msmnt (Feet) :=
                        Positive_Real (Integer (Msmnt (Feet) - 0.5) mod 3);
            end if;
            if Msmnt (Yards) >= 1760.0 then
                 Msmnt (Miles) := Msmnt (Miles) +
                        Positive_Real (Integer (Msmnt (Yards) - 0.5) / 1760);
                Msmnt (Yards) :=
                        Positive_Real (Integer (Msmnt (Yards) - 0.5) mod 1760);
            end if;

                Result.Inches := Msmnt (Inches);
                Result.Feet   := Msmnt (Feet);
                Result.Yards  := Msmnt (Yards);
                Result.Miles  := Msmnt (Miles);

            return Result;

        end Adjusted_Measurements;

-- Converts a number of foreign units to English units.  Gets conversion
-- information by calling Look_Up_Conversion_Values, and calls
-- Adjusted_Measurements to adjust the values in the resulting English
-- measurement.

        function Conversion
           (Foreign_Unit_Name : Foreign_Unit_Name_Type;
            Number_of_Units   : Positive_Real)
            return Measurement_Type is

            Conversion_Factor  : Positive_Real;
            Conversion_Unit    : English_Units_Type;
```

```
    begin     -- Conversion

        Look_Up_Conversion_Values
                (Foreign_Unit_Name, Conversion_Unit, Conversion_Factor);

        return Adjusted_Measurements
                (Number_of_Units * Conversion_Factor, Conversion_Unit);

    end Conversion;

end English_Measurements;
```

EXERCISE 1.2
INPUT/OUTPUT

Objective

To review Text_IO and to introduce the other Ada I/O facilities.

Tutorial

There are four I/O packages defined in Ada. They are Text_IO, Sequential_IO, Direct_IO, and Low_Level_IO. Text_IO is used to perform input/output on text files. Sequential_IO is used for input/output on sequential access files. Direct_IO is used for input/output on direct access files. Low_Level_IO is used for input/output operations on a physical device.

In Ada, I/O (except Low_Level_IO) is performed on files of predefined type File_Type. When talking about files, we will refer to the internal file (the file object inside the program) or the external file (the physical file).

The most important aspect of I/O to keep in mind is that much of I/O is implementation dependent. For example, when the file is an interactive input file, how the implementation determines the end of the file will vary from system to system. Some systems may assume that the end of the file is signaled by a code (and which code may vary from system to system), while others may wait for some specified unit of time and when no data is received, will assume the end of the file has been reached. Again, keep in mind that I/O is very likely to change from implementation to implementation.

Recall from the Ada Primer, that the package Text_IO contains the following basic procedures and functions for controlling files in Ada:

- CREATE : Establishes a new external file and associates this external file with the given internal file. The internal file is left open.
- OPEN : Associates the given internal file with an existing external file.
- CLOSE : Severs the association between the internal and external file.

DELETE : Deletes the external file associated with the given internal file and closes the internal file.

RESET : Resets the given file so that reading from or writing to the file starts at the beginning of the file.

MODE : Returns the current mode of a given file.

NAME : Returns the name of the external file associated with the given internal file.

FORM : Returns the form of the external file associated with the given internal file.

IS_OPEN : Returns True when the given internal file is open, False when otherwise.

These subprograms also exist in Sequential_IO and Direct_IO and have the same effect.

Text_IO contains other subprograms as well. Some of them manipulate the default files. The default files are system files which the Text_IO subprograms use when no other file is explicitly given in the subprogram call. At the start of program execution, the standard input file and the standard output file are open and they are the current input file and the current output file, respectively. The commands to manipulate these files are:

SET_INPUT : Sets the current default input file to the given file.

SET_OUTPUT : Sets the current default output file to the given file.

STANDARD_INPUT : Returns the standard input file.

STANDARD_OUTPUT : Returns the standard output file.

CURRENT_INPUT : Returns the current default input file.

CURRENT_OUTPUT : Returns the current default output file.

Other Text_IO subprograms operate on, or obtain information from, input text files. They are:

SKIP_LINE : Reads and discards characters until the end of a line is reached.

SKIP_PAGE : Reads and discards lines until the end of a page is reached.

END_OF_LINE : Returns True if currently positioned at the end of a line.

END_OF_PAGE : Returns True if currently positioned at the end of a page.

END_OF_FILE : Returns True if currently positioned at the end of a file.

The Text_IO subprograms which only apply to output text files are:

SET_LINE_LENGTH : Sets the maximum line length for the file.

SET_PAGE_LENGTH : Sets the maximum page length for the file.

LINE_LENGTH : Returns the maximum line length for the file.

PAGE_LENGTH : Returns the maximum page length for the file.

NEW_LINE : Writes an end of line marker to the file.

NEW_PAGE : Writes an end of page marker to the file.

The Text_IO subprograms which apply to input or output files are:

SET_COL : (For input files) Reads and discards characters until the column count equals the given parameter value.

(For output files) Outputs blank characters and end of line markers until the current column count equals the given parameter value.

SET_LINE : (For input files) Reads and discards lines until the line count equals the given parameter value.

(For output files) Output end of line markers and end of page markers until the current line count equals the given parameter value.

COL : Returns the value of the file's column counter.

LINE : Returns the value of the file's line counter.

PAGE : Returns the value of the file's page counter.

Recall that Text_IO uses Get to receive information from a file and Put to send information to a file. Get and Put for values of type Character and String are automatic in Text_IO, that is to say, no instantiation is necessary. The capability for Get and Put of other types is in generic packages contained in Text_IO. These generic packages must first be instantiated for the type (see Exercise 3.3 for a review of instantiation). These generic packages are:

Integer_IO
Fixed_IO
Float_IO
Enumeration_IO

Recall, also, that the only composite type that Text_IO operates on is String. Other composite types must be operated on by individual components.

Sequential_IO and Direct_IO have no formatting capability. They transfer data as is, bit by bit. This makes them appropriate for working files that will never be examined by human beings. Sequential_IO is a generic package which, when instantiated with a type, performs I/O for that type on sequential access files. The mode of these sequential access files is either In_File or Out_File. Direct_IO is also a generic package. When it is instantiated with a type, it performs I/O for that type on direct (random) access files. The mode of direct access files can be In_File, Out_File, or Inout_File.

The possible commands for Sequential_IO (aside from CREATE, OPEN, CLOSE, DELETE, RESET, MODE, NAME, FORM, and IS_OPEN, as mentioned above) are:

READ : Returns the next value in the specified file.

WRITE : Stores the given value at the end of the specified file.

END_OF_FILE : Returns True if no more elements can be read from the file.

The possible commands for Direct_IO (aside from those mentioned above) are:

READ : Returns the value that is located at the position specified by the current index of the file.

WRITE : Stores the given value in the location specified by the value of the current index.

SET_INDEX : Sets the current index to a specified value.

INDEX : Returns the value of the current index.

SIZE : Returns the current size of the external file associated with the specified internal file.

END_OF_FILE : Returns True if the current index exceeds the size of the external file.

And quickly, the possible commands available in Low_Level_IO are:

SEND_CONTROL : Sends control information to a physical device.

RECEIVE_CONTROL : Requests control information from a physical device.

So much of Low_Level_IO is implementation-dependent that a meaningful example cannot be given. Although much of Sequential_IO and Direct_IO is implementation-dependent also, enough is defined that meaningful examples can be given. The following examples illustrate how Direct_IO and Sequential_IO could be used.

Suppose a record type represents the record layout of a tape file. Performing I/O on this file could be done using Sequential_IO as in the following:

```
with Sequential_IO;
procedure Process_Tape is

    type Data_Rec is
        record
            Part_Name : String (1 .. 10);
            Part_Size : Float;
            In_Stock  : Boolean;
        end record;

    package Tape_IO is new Sequential_IO (Data_Rec);

    Tape_File : Tape_IO.File_Type;
    Input_Rec : Data_Rec;

    procedure Process_Record (Rec : in Data_Rec) is separate;

begin  -- Process_Tape

    Tape_IO.Open (File => Tape_File,
                  Mode => Tape_IO.In_File,
                  Name => "Master_Tape");

    while not Tape_IO.End_Of_File (Tape_File)
    loop
        Tape_IO.Read (Tape_File, Input_Rec);
        Process_Record (Input_Rec);
    end loop;

    Tape_IO.Close (Tape_File);

end Process_Tape;
```

Note that Sequential_IO is a library unit which must be imported (like Text_IO) before it can be used. Note also that the only use that can be made of Sequential_IO is to instantiate it. After the instantiation, the newly created package is used to perform the I/O operations.

Suppose our external file contains two kinds of records, header records and data records. Assume the header record is followed by an arbitrary number of data records, and this arbitrary number is stored in the header record. A possible approach for processing this file follows.

```
with Sequential_IO;
procedure Process_Data_File is

    type Header_Rec is
        record
            Data_Class : String (1 .. 5);
            Rec_Count  : Natural;
        end record;

    type Data_Rec is
        record
            Name     : String (1 .. 10);
            Size     : Float;
            In_Stock : Boolean;
        end record;

    procedure Process_Data (Data : in Data_Rec) is separate;

    package Header_IO   is new Sequential_IO (Header_Rec);
    package Data_Rec_IO is new Sequential_IO (Data_Rec);

    File_Name        : constant String := "Data_File";
    File1            : Header_IO.File_Type;
    File2            : Data_Rec_IO.File_Type;
    Input_Header_Rec : Header_Rec;
    Input_Data_Rec   : Data_Rec;

begin -- Process_Data_File

    Header_IO.Open   (File1, Header_IO.In_File,   File_Name);
    Data_Rec_IO.Open (File2, Data_Rec_IO.In_File, File_Name);

    while not End_Of_File (File1)
    loop

        Header_IO.Read   (File1, Input_Header_Rec);

        for I in 1 .. Input_Header_Rec.Rec_Count
        loop
            Data_Rec_IO.Read (File2, Input_Data_Rec);
            Process_Data     (Input_Data_Rec);
        end loop;

    end loop;

    Header_IO.Close   (File1);
    Data_Rec_IO.Close (File2);

end Process_Data_File;
```

Note that this example is implementation-dependent in that not all implementations will support having two internal files associated with the same external file.

However, this consideration notwithstanding, the example still illustrates an interesting feature; specifically, that the operations apply only for data of a specific type. Each type has a separate instantiation which creates a separate subprogram for Create, Open, Close, etc.

This holds true for direct access files also. The main differences between Direct_IO and Sequential_IO are that Direct_IO has the capability to access files in non-sequential order, and that it can both read and write, alternately, to the file (i.e., when the mode is Inout_File). Again, note how highly implementation dependent this is. Implementations that are targeted for machines which do not support random access files certainly will not support Direct_IO.

The following illustrates a possible use of Direct_IO on a system that supports it.

Suppose a file containing inventory information is stored in a random access file. Further imagine that the file is continually being accessed (to find cost information) and updated (to adjust the quantity).

```
with Direct_IO;
procedure Inventory_Control is

    subtype Name is String (1 .. 10);
    type Money is delta 0.01 range 0.0 .. 1.0E6;
    type Data_Rec is
        record
            Item_Name     : Name;
            Cost          : Money;
            Price         : Money;
            Profit        : Money;
            In_Stock      : Natural;
            Reorder_level : Natural;
        end record;

    package Data_IO is new Direct_IO (Data_Rec);
    use Data_IO;
```

```
        Inventory_File : File_Type;
        Inventory_Name : constant String := "Master_Inventory.Dat";
        Inventory_Rec  : Data_Rec;

        -- Returns the index value of the given item.
        -- Assume these are stored elsewhere

        function Item_Index (Item : Name)
                            return Positive_Count is separate;

        function Item_Price (Item : Name) return Money is

            Index : Positive_Count;

        begin -- Item_Price

            Index := Item_Index (Item);
            Read (Inventory_File, Inventory_Record, Index);
            return Inventory_Record.Price;

        end Item_Price;

        procedure Update_Inventory (Item     : in Name;
                                    Quantity : in Integer) is

            -- Note that Quantity can be positive (when inventory is
            -- added) or negative (when inventory is sold)

            Index : Positive_Count;

        begin -- Update_Inventory

            Index := Item_Index (Item);
            Read (Inventory_File, Inventory_Record, Index);
            Inventory_Record.In_Stock := Inventory_Record.In_Stock
                                         + Quantity;
            Write (Inventory_File, Inventory_Record, Index);

        end Update_Inventory;
        .
        .

    begin -- Inventory_Control

        Open (Inventory_File, Inout_File, Inventory_Name);
        .
        .

        Close (Inventory_File);

    end Inventory_Control;
```

Of primary interest are the subprograms Item_Price and Update_Inventory. They contain the actual reads and writes. Note specifically that Update_Inventory both reads from, and writes to, the file. This is possible because the file was opened with mode Inout_File. Also note the use of the index, which identifies the position of the record to be read or written. This method of identifying the location of the record to be read or written is the essence of performing I/O on direct access files.

A final note on Ada I/O -- Each I/O feature is capable of raising an I/O exception. The I/O exceptions are defined in the package IO_Exceptions which Text_IO, Sequential_IO, and Direct_IO "with"s. These exceptions are:

STATUS_ERROR : Raised when an operation is performed on an unopen file.

MODE_ERROR : Raised when an operation is performed on a file with an incorrect mode, i.e., writing to a file that was opened with mode In_File.

NAME_ERROR : Raised when an illegal name is given as the name of an external file.

USE_ERROR : Raised when an implementation is unable to support a request, i.e., open a file associated with the card reader with mode outfile.

DEVICE_ERROR : Raised when an operation cannot be completed because of a malfunction in the hardware.

END_ERROR : Raised when an attempt is made to read past the end of the file.

DATA_ERROR : Raised when an element read cannot be interpreted as a value of the required type.

LAYOUT_ERROR : Raised when Col, Line, or Page returns a value greater than Count'Last or when an element is written which has a length greater than the maximum line length.

Problem

Modify the package English_Measurements from Exercise 1.1 to allow new foreign measurements to be added to the conversion table.

Note: The conversion values can no longer be hard coded into the program because permanent changes to hard coded values require recompilation. Even allowing room in the table for additions will not solve the problem because conversions added at execution time will be lost when the program completes execution. The conversions must be stored externally, that is to say, in an external file, so that the new measurement conversions are permanently added to the table. Specify what changes need be made.

Solution and Discussion

We start with a package that contains:

* the types for the different units of measure,

* and a function which performs the conversions.

The first step towards our goal is to add the procedure Add_Conversion_To_Table to the package specification of English_Measurements. This allows any user of the package to add new conversions. It must be included in the package specification because the actual implementation of the table is hidden from the user. Note that the actual information to add is passed as parameters. We also need some mechanism to signal the user when the table is full. We use an exception, Table_Full, for this. The following shows our modified package specification.

```
package English_Measurements is

    type Positive_Real is digits 5 range 0.0 .. 5000.0;

    type English_Units_Type is (Inches, Feet, Yards, Miles);

    type Measurement_Type is
        record
            Inches : Positive_Real range 0.0 .. 12.0;
            Feet   : Positive_Real range 0.0 .. 3.0;
            Yards  : Positive_Real range 0.0 .. 1760.0;
            Miles  : Positive_Real;
       end record;

    subtype Foreign_Unit_Name_Type is String (1 .. 10);

    Measure_Not_Found : exception;
    Table_Full        : exception;

    function Conversion
        (Foreign_Unit_Name : Foreign_Unit_Name_Type;
         Number_of_Units   : Positive_Real)
        return Measurement_Type;

    procedure Add_Conversion_To_Table
        (Name         : in Foreign_Unit_Name_Type;
         English_Unit : in English_Units_Type;
         Factor       : in Positive_Real);

end English_Measurements;
```

Now we change the package body to implement Add_Conversion_To_Table. The first step is to delete the initialization of the table from the package, define an external file to contain these values, and modify the package body so that it loads the table whenever the package is "with"ed. The following illustrates these changes to the body of English_Measurements.

```
with Sequential_IO;
package body English_Measurements is

    type Foreign_Unit_Type is
        record
            Unit_Name         : Foreign_Unit_Name_Type;
            Conversion_Unit   : English_Units_Type;
            Conversion_Factor : Positive_Real;
        end record;

    type Conversion_Table_Type is array (Natural range<>)
        of Foreign_Unit_Type;

    Conversion_Table : Conversion_Table_Type (1 .. 14);

    package Measure_IO is new Sequential_IO (Foreign_Unit_Type);

    Table_File         : Measure_IO.File_Type;
    Measure_Table_Name : constant String := "Measurement_Table.Dat";

    procedure Add_Conversion_To_Table
        (Name         : in Foreign_Unit_Name_Type;
         English_Unit : in English_Units_Type;
         Factor       : in Positive_Real)
        is separate;

    procedure Look_Up_Conversion_Values
        (Foreign_Unit_Name : in  Foreign_Unit_Name_Type;
         Conversion_Unit   : out English_Units_Type;
         Conversion_Factor : out Positive_Real)
        is separate;

    function Adjusted_Measurements
        (Quantity : Positive_Real;
         Unit     : English_Units_Type)
        return Measurement_Type is separate;

    function Conversion
        (Foreign_Unit_Name : Foreign_Unit_Name_Type;
         Number_of_Units   : Positive_Real)
        return Measurement_Type is separate;
```

```
begin -- English_Measurements

    Measure_IO.Open (File => Table_File,
                     Mode => Measure_IO.In_File,
                     Name => Measure_Table_Name);
    for Index in 1 .. 14
    loop
        Measure_IO.Read (Table_File, Conversion_Table(Index));
    end loop;
    Measure_IO.Close (Table_File);

end English_Measurements;
```

This is fine, except that it does not allow the table to grow as new conversions are added. How do we do this? Well, we can use a variable to specify the size of the table and allocate the table to be X units greater than needed at the present time. This will allow X new conversions to be added for any given run of the system. We will choose 10 as the value of X for our solution.

But how do we know the current table size at the start of execution? Here a design decision must be made. Do we count the records in the external file? No, very inefficient. Could we store the size as the first element in the file? Yes we could, but there could be a problem because there would be data items of two different types in the file which requires two different internal files to be associated with the external file, and as you recall, not all implementations will support this. OK, so could we store it some place else? Yes, but this is unnecessary. Since what we really want is the size of the external file, and as you recall, Direct_IO has a command, Size, which returns exactly this, it makes sense to use Direct_IO instead of Sequential_IO.

Now we can define a function, called Table_Size, which opens the file, determines its size, closes the file, and returns the size.

A problem arises, however, because of the placement of this function. It must precede the declaration of the table, and the rules for declarative regions specify that all type and object declarations must precede any subprogram bodies. The following shows a possible solution.

```
with Direct_IO;
package body English_Measurements is

    type Foreign_Unit_Type is
        record
            Unit_Name          : Foreign_Unit_Name_Type;
            Conversion_Unit    : English_Units_Type;
            Conversion_Factor  : Positive_Real;
        end record;

    type Conversion_Table_Type is array (Natural range <>)
        of Foreign_Unit_Type;

    package Measure_IO is new Direct_IO (Foreign_Unit_Type);

    Table_File         : Measure_IO.File_Type;
    Measure_Table_Name : constant String := "Measurement_Table.Dat";

    function Table_Size return Natural is

        Size : Natural;

    begin -- Table_Size

        Measure_IO.Open  (Table_File, Measure_IO.In_File, Table_Name);
        Size := Measure_IO.Size (Table_File);
        Measure_IO.Close (Table_File);
        return Size;

    end Table_Size;

    package Measure_Table is
        Size : Natural := Table_Size;
        Conversion_Table :
            Conversion_Table_Type (1 .. Size + 10);
    end Measure_Table;

    use Measure_Table;
```

```
procedure Add_Conversion_To_Table
    (Name         : in Foreign_Unit_Name_Type;
     English_Unit : in English_Unit_Type;
     Factor       : in Positive_Real)
    is separate;

procedure Look_Up_Conversion_Values
    (Foreign_Unit_Name : in  Foreign_Unit_Name_Type;
     Conversion_Unit   : out English_Units_Type;
     Conversion_Factor : out Positive_Real)
    is separate;

function Adjusted_Measurements
    (Quantity : Positive_Real;
     Unit     : English_Units_Type)
    return Measurement_Type is separate;

function Conversion
    (Foreign_Unit_Name : Foreign_Unit_Name_Type;
     Number_of_Units   : Positive_Real)
    return Measurement_Type is separate;

begin -- English_Measurements

    Measure_IO.Open (Table_File,
                     Measure_IO.In_File,
                     Measure_Table_Name);
    for Index in 1 .. Size
    loop
        Measure_IO.Read (Table_File, Conversion_Table(Index), Index);
    end loop;
    Measure_IO.Close (Table_File);

end English_Measurements;
```

This solution puts the table declaration in a nested package specification which is allowed to follow subprogram bodies. The package specification is subsequently "use"d to allow direct visibility of the table to all the following subprograms which need it.

Now only Add_Conversion_To_Table remains to be written. Functionally, this subprogram must check that the conversion can be added to the internal table, open the external file, write the new information to the end of the file, close the external file, and then add the information to the internal file. This subprogram follows.

```
separate (English_Measurements)
procedure Add_Conversion_To_Table
        (Name         : in Foreign_Unit_Name_Type;
         English_Unit : in English_Units_Type;
         Factor       : in Positive_Real) is

    New_Measure : Foreign_Unit_Type := (Unit_Name         => Name,
                                        Conversion_Unit   => English_Unit,
                                        Conversion_Factor => Factor);

begin -- Add_Conversion_To_Table

    -- if room in table

    if Size < Conversion_Table'Last then

        --  Update External Table

        Size := Size + 1;
        Measure_IO.Open  (Table_File,
                          Measure_IO.Inout_File,
                          Measure_Table_Name);
        Measure_IO.Write (Table_File, New_Measure, Size);
        Measure_IO.Close (Table_File);

        --  Update Internal File

        Conversion_Table (Size) := New_Measure;

    else

        raise Table_Full;

    end if;

end Add_Conversion_To_Table;
```

The major point to note is that Add_Conversion_To_Table is easy to implement using Direct_IO. Had we continued with Sequential_IO, we would have had trouble adding the data to the end of the file. Positioning the current index at the end of the file is practically impossible in Sequential_IO. Positioning the index at the end of the file requires that the whole file be rewritten; then the new material can be appended. That solution is such a poor approach that it, by itself, justifies the switch to Direct_IO.

The complete solution (except for the bodies of Adjusted_Measurements, Look_Up_Conversion_Factors, and Conversion which do not change from Exercise 1.1) follows:

```
package English_Measurements is

        type Positive_Real is digits 5 range 0.0 .. 5000.0;

        type English_Units_Type is (Inches, Feet, Yards, Miles);

        type Measurement_Type is
            record
                Inches : Positive_Real range 0.0 .. 12.0;
                Feet   : Positive_Real range 0.0 .. 3.0;
                Yards  : Positive_Real range 0.0 .. 1760.0;
                Miles  : Positive_Real;
            end record;

        subtype Foreign_Unit_Name_Type is String (1 .. 10);

        Measure_Not_Found : exception;
        Table_Full        : exception;

        function Conversion
            (Foreign_Unit_Name : Foreign_Unit_Name_Type;
             Number_of_Units   : Positive_Real)
            return Measurement_Type;

        procedure Add_Conversion_To_Table
            (Name         : in Foreign_Unit_Name_Type;
             English_Unit : in English_Units_Type;
             Factor       : in Positive_Real);

end English_Measurements;

with Direct_IO;
package body English_Measurements is

    type Foreign_Unit_Type is
        record
            Unit_Name         : Foreign_Unit_Name_Type;
            Conversion_Unit   : English_Units_Type;
            Conversion_Factor : Positive_Real;
        end record;

    type Conversion_Table_Type is array (Natural range <>)
        of Foreign_Unit_Type;

    package Measure_IO is new Direct_IO (Foreign_Unit_Type);

    Table_File         : Measure_IO.File_Type;
    Measure_Table_Name : constant String := "Measurement_Table.Dat";
```

```
    function Table_Size return Natural is

        Size : Natural;

    begin -- Table_Size

        Measure_IO.Open  (Table_File, Measure_IO.In_File, Table_Name);
        Size := Measure_IO.Size (Table_File);
        Measure_IO.Close (Table_File);
        return Size;

    end Table_Size;

    package Measure_Table is
        Size : Natural := Table_Size;
        Conversion_Table : Conversion_Table_Type (1 .. Size + 10);
    end Measure_Table;

    use Measure_Table;

    procedure Add_Conversion_To_Table
            (Name         : in Foreign_Unit_Name_Type;
             English_Unit : in English_Units_Type;
             Factor       : in Positive_Real)
        is separate;

    procedure Look_Up_Conversion_Values
        (Foreign_Unit_Name : in  Foreign_Unit_Name_Type;
         Conversion_Unit   : out English_Units_Type;
         Conversion_Factor : out Positive_Real)
        is separate;

    function Adjusted_Measurements
        (Quantity : Positive_Real;
         Unit     : English_Units_Type)
        return Measurement_Type is separate;

    function Conversion
        (Foreign_Unit_Name : Foreign_Unit_Name_Type;
         Number_of_Units   : Positive_Real)
        return Measurement_Type is separate;

begin -- English_Measurements

    Measure_IO.Open (Table_File, Measure_IO.In_File, Measure_Table_Name);
    for Index in 1 .. Size
    loop
        Measure_IO.Read (Table_File, Conversion_Table(Index), Index);
    end loop;
    Measure_IO.Close (Table_File);

end English_Measurements;
```

```
separate (English_Measurements)
procedure Add_Conversion_To_Table
    (Name         : in Foreign_Unit_Name_Type;
     English_Unit : in English_Units_Type;
     Factor       : in Positive_Real) is

    New_Measure : Foreign_Unit_Type := (Unit_Name         => Name,
                                         Conversion_Unit   => English_Unit,
                                         Conversion_Factor => Factor);

begin -- Add_Conversion_To_Table

    -- if room in table

    if Size < Conversion_Table'Last then

        --  Update External Table

        Size := Size + 1;
        Measure_IO.Open  (Table_File,
                          Measure_IO.Inout_File,
                          Measure_Table_Name);
        Measure_IO.Write (Table_File, New_Measure, Size);
        Measure_IO.Close (Table_File);

        --  Update Internal File

        Conversion_Table (Size) := New_Measure;

    else

        raise Table_Full;

    end if;

end Add_Conversion_To_Table;
```

Note that the expansion margin of 10 that we chose is arbitrarily confining. Later, in Exercise 2.2, we will address programming techniques that can be used to make the table essentially unbounded.

CHAPTER 2

BASIC DATA STRUCTURES

EXERCISE 2.1

SETS: REPRESENTATION, USE, AND BASIC OPERATIONS

Objective

To illustrate an implementation of set abstraction through arrays of Booleans.

Tutorial

This tutorial is divided into two sections. The first briefly reviews set theory. The second part of the tutorial discusses an implementation of sets in Ada. It should be noted that the first section uses traditional mathematical notations; any resemblance with the notation of Ada or any other programming languages is to be regarded as accidental.

Set Theory

Sets are used to represent the notion of a collection of unordered data items. Typically, these items are grouped together in a set because they share some common properties. For example, the letters of the alphabet are a set. The property they share is that they are the atomic units of written language. This set could be written as

$$\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z\}$$

where the elements of the set are listed and enclosed by curly braces. Because this set is intended to represent the letters of the alphabet and not their graphical representation, no distinction is made in the set between upper and lower case characters. There is no ordering to the elements of a set. They are related only insofar as they belong to the same set. Thus, the above set is identical to the one below:

$$\{B, E, Z, O, A, C, D, F, X, M, N, P, R, U, V, L, Y, G, H, K, J, W, Q, T, I, S\}$$

In discussing sets, it is often useful to refer to a set by name, without enumerating the elements every time. This process of naming the set is straightforward. For example, the set Vowels can be named as:

Vowels = {A, E, I, O, U}

and the set G as:

G = {G, R, A, N, D, C, Y, O}

A set alone is of limited usefulness; one needs to define operations on the set to use it productively. The six most frequently used operations, subsetting, intersection, union, difference, complement, and insertion will be defined in this tutorial.

When each element of one set is also an element of a second set, then the first set is considered to be a subset of the second. To illustrate this point, consider the set Vowels defined earlier. Each element of this set also belongs to the set containing the letters of the alphabet. Therefore, the set of vowels is a subset of the set of alphabet letters. A subset and its "parent" set have all the elements of the subset in common.

A related concept is that of intersection; the set of elements which any two or more sets have in common. Consider the following set G of distinct letters forming the words "Grand Canyon:"

G = {G, R, A, N, D, C, Y, O}

Note how no elements of the set are duplicated. The intersection of this set with the set of vowels described earlier may be written as:

{G, R, A, N, D, C, Y, O} * {A, E, I, O, U}

or more simply,

Vowels * G

which is the set:

{A, O}

The symbol for intersection is sometimes written as $\cap$. Because this symbol is not part of the Ada character set, and because there are certain mathematical analogies between intersection of sets and multiplication of numbers, the equally acceptable multiplication symbol, *, will be used instead.

The intersection of two sets is not guaranteed to contain elements. When there are no elements in common between two sets, then that intersection is said to be the empty set. Consider the intersection of the set of consonants with the set of vowels:

{B, C, D, F, G, H, J, K, L, M, N, P, Q, R, S, T, V,
W, X, Y, Z} *
{A, E, I, O, U}

This set has no elements and is known as the empty or null set:

{ }

Another operation performed on sets is combining two or more sets into a single set. This process is called union, and the resulting set contains all the elements of both sets, listing the common elements only once. An example is once again in order. Consider the union of the set of letters in the names "Bright Angel Trail" and "Plateau Point:"

{B, R, I, G, H, T, A, N, E, L} +
{P, L, A, T, E, U, O, I, N}

The union set is

{B, R, I, G, H, T, A, N, E, L, P, U, O}

The union operator is the symbol $\cup$. For the reasons given above for intersection, we shall use the addition symbol, +, instead.

Another way of comparing the elements of two sets is to find the set whose elements are all members of one set but not of another. This operation, called set difference, is represented with the symbol -. The set difference

{B, R, I, G, H, T, A, N, E, L, P, U, O} -
{G, R, A. N, D, C, Y, O}

is the set

{B, I, H, T, E, L, P, U}

Note that the set following the - may contain elements not in the set preceding the -. Reversing the order of the operands,

{G, R, A, N, D, C, Y, O} -
{B, R, I, G, H, T, A, N, E, L, P, U, O}

yields a different result:

{D, C, Y}

In set theory, the universe is defined as the set of all elements pertinent to the problem. For the union, intersection, and difference operations illustrated above, the universe set could be the set of the letters of the alphabet. Another equally valid universe set for these examples would be the set of all printable ASCII characters. It is important to identify the universe set because it provides the context for the complement operation. The complement of a set S is the set of all elements of the universe not in S. The complement, denoted by the symbol, ', of the set of vowels

{A, E, I, O, U} '

is the set of consonants

{B, C, D, F, G, H, J, K, L, M, N, P, Q, R, S, T,
V, W, X, Y, Z}

where the universe is defined to be the set of letters of the alphabet. Set complement may also be denoted by drawing a line above the set, as in

$\overline{\text{Vowels}}$

The complement of any set S relative to some universe U is always identical to the set difference U - S.

The insertion operation is used to add a new element into a set. It is basically a special case of the union operation, such that one operand is a set and the other operand is

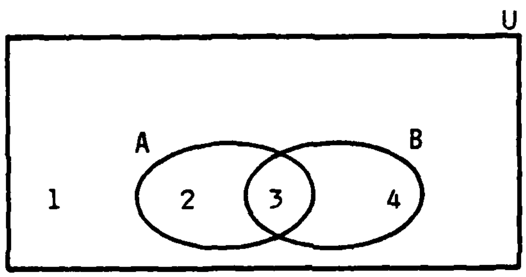
the element which is to be added to that set. Because union operates on sets, the element to be inserted must in fact also be contained inside a set. To insert the element P into the set G, we write:

G + {P}

The resulting set is:

{G, R, A, N, D, C, Y, O, P}

To recapitulate, sets are sets of items belonging to some universe set. Relationships among sets include subset, intersection, union, difference, complement, and insertion. Pictorially, we have



where A and B are subsets of some universe set U. The four regions identified in the diagram are described as follows, illustrating the various set operations.

Region 1:	(A + B)', also A' * B'
Region 2:	A - B
Region 3:	A * B
Region 4:	B - A
Regions 2 and 3:	A
Regions 3 and 4:	B
Regions 2, 3, and 4:	A + B
Regions 2 and 4:	(A + B) - (A * B)
Regions 1 and 2:	B'
Regions 1 and 4:	A'

Implementation of Sets

Having discussed sets and their basic operations as an abstraction, let us now examine their Ada implementation. Starting with the notion of a set, an appropriate data structure is an array of Booleans indexed by

the elements of the universe set, such that if the element is a member of some set, then the corresponding array component has the value True. If it is not an element of this set, then the corresponding array component has the value False. Consider the following set of parts as the universe set, U:

```
U = {Nut, Bolt, Screw, Philips_Screw, Nail, Hook,
     Clamp, Wire, Switch, Washer}
```

In Ada, the elements of this universe can be defined by an enumeration type declaration:

```
type Parts_Type is
   (Nut, Bolt, Screw, Philips_Screw, Nail, Hook,
    Clamp, Wire, Switch, Washer);
```

A given assembly needs a set of spare parts. This set is defined through the following array type definition:

```
type Assembly_Set is array (Parts_Type) of Boolean;
```

The set of parts for a particular object is created by assigning a value of True to the components indexed by the set elements. Specifically, as we see below, the object Assembly_1, representing the set

{Nut, Screw}

has the components indexed by Nut and Screw set to True and all others set to false. Likewise, the set of auxiliary parts for Assembly_2 has elements Philips_Screw, Wire, and Switch. The parts list for Assembly_3 consists of Screw, Bolt, Washer, Wire and Switch. Note how using named aggregates and the others construct in defining the set of parts provides clarity.

```
Assembly_1 : Assembly_Set :=    --   {Nut, Screw}
                 Assembly_Set'(Nut  | Screw => True,
                               others       => False);

Assembly_2 : Assembly_Set :=    --   {Philips_Screw, Wire, Switch}
                 Assembly_Set'(Philips_Screw | Wire | Switch
                                     => True,
                               others => False);
```

```
Assembly_3 : Assembly_Set :=    --    Screw, Bolt, Washer, Wire,
                                --    Switch
             Assembly_Set'(Screw | Bolt | Washer | Wire |
                           Switch
                                  => True,
                           others => False);
```

Notice that the aggregate appears in a qualified expression. The language rules concerning aggregates require a qualified expression here to identify which components should be given the value False. Although there are some contexts in which qualification of this aggregate is not required, when a named aggregate has an **others** choice, it is usually necessary to qualify the aggregate with the name of a constrained array type, and it is always safe to do so.

Recall that the array indices represent the elements of the universe set; the corresponding component values of type Boolean indicate whether or not a particular element of the universe is in the set represented by a given array. Thus the universe set and the empty set look like this:

```
Universe_Set : constant Assembly_Set :=
                              (Parts_Type => True);
Empty_Set    : constant Assembly_Set :=
                              (Parts_Type => False);
```

Set operations can be implemented using the logical operators **and**, **or**, and **not**. When applied to one-dimensional arrays of Boolean components, these operations are applied on a component-by-component basis, producing another such array holding the component-by-component Boolean results. For example:

```
(True, False, True,  False) and (True, False, False, True)
```

yields

```
(True, False, False, False)
```

The complement of a set is defined to be all elements of the universe set not in that set. "Not" is the key word here! The array components marked False are in the complement set, while the array components marked True are not in the complement set. The logical operator not accomplishes this inversion:

```
not Assembly_1  -- {Bolt, Philips_Screw, Nail, Hook, Clamp,
                --  Wire, Switch, Washer}
```

Intersection is defined as those elements both in one set and another set. The logical operator and implements this relationship:

```
Assembly_1 and Assembly_2  -- the empty set:{ }
Assembly_1 and Assembly_3  --  {Screw}
```

The union of two sets is the set whose elements are either in A, or in B, or in both. The logical operator or is used to compute union:

```
Assembly_2 or Assembly_3  --  {Philips_Screw, Screw, Bolt,
                          --   Washer, Wire, Switch}
```

Set difference is the set whose elements are in the first set and not in the second set, as shown in the following expression:

```
Assembly_1 and not Assembly_3  --  {Nut}
```

The subset relationship is defined between two sets if all the elements of the first set are also members of the second set; in other words, if the intersection of these two sets is identical to the first set. The following expression, which represents this relationship, evaluates to True if Assembly_2 is a subset of Assembly_3, False otherwise:

```
(Assembly_2 and Assembly_3) = Assembly_2  -- False: Assembly_2
                                          -- is not a subset
                                          -- of Assembly_3
```

Alternatively, the existence of a subset may be computed as follows: if the difference between set A and set B is the empty set, then A is a subset of B. Thus the previous example could also be expressed as:

```
(Assembly_2 and not Assembly_3) = Empty_Set  -- False
```

The insertion operation is defined as adding an element E to a set S. In the implementation discussed, membership in a set is denoted by the indexed component corresponding to some element in the array S having the value True. To insert an element, therefore, requires setting the indexed component corresponding to the element to True. If Assembly_1 is to be modified by inserting a clamp into the set, it would be done as follows:

```
Assembly_1 (Clamp) := True;
```

So far, we have discussed the implementation of sets as arrays of Booleans and of set operations as combinations of the Boolean operators. We must now address the problem of structuring our set implementation in such a way that a programmer can use sets at the "set-level" of abstraction. There are two aspects to this problem: the first is the issue of how to express sets at a set-level of abstraction; the second concern is how to present this set-level abstraction to the user.

The first issue may be reworded as follows: the user should be able to express the union of sets A and B as

```
A + B
```

rather than

```
A or B
```

The logical Boolean operators on arrays reveal the implementation and represent an inappropriate level of abstraction for expressing set relationships. Using the arithmetic operators discussed earlier in the tutorial would enhance the clarity of the code. Because logical Boolean operations are not predefined for the arithmetic operators, these must be overloaded. Alternatively, a reader may prefer to use function names such as Union, Intersection, and Difference. Functions can be defined for subset and complement (the apostrophe symbol, ', is not a legal Ada operator and thus may not be overloaded). A procedure is defined for insertion of an element into a set.

The second issue described above relates to the concept of packaging. Recall that a package serves to group together logically related entities, such as a data type and the operations defined on that type. Furthermore, the package concept enforces a separation of concerns between the specification of the interface (package specification) and the implementation of that interface (package body). Thus using a package would solve the second problem of how to present sets to the user. The data type representing sets, i.e. the array of Booleans, and the operations defined on sets, i.e. union, intersection, difference, complement, subset, and insertion are specified in the package specification. The implementation of these operations is hidden in the package body. The specification of this package is:

```
package Assembly_Set_Package is

    -- Elements of set:

    type Parts_Type is
        (Nut, Bolt, Screw, Philips_Screw, Nail, Hook,
         Clamp, Wire, Switch, Washer);

    -- Set type definition

    type Assembly_Set is array (Parts_Type) of Boolean;

    -- The following constants are defined for convenience

    Universe_Set : constant Assembly_Set :=
                                        (Parts_Type => True);
    Empty_Set    : constant Assembly_Set :=
                                        (Parts_Type => False);

    -- Union

    function "+" (Set_1, Set_2 : Assembly_Set)
                 return Assembly_Set;

    -- Intersection

    function "*" (Set_1, Set_2 : Assembly_Set)
                 return Assembly_Set;

    -- Difference

    function "-" (Set_1, Set_2 : Assembly_Set)
                 return Assembly_Set;
```

```
-- Complement

 function Complement (Set : Assembly_Set) return Assembly_Set;

 -- Subset

 function Is_Subset (Set_1, Of_Set_2 : Assembly_Set)
                   return Boolean;

 -- Insertion

 procedure Insert (Element  : in Parts_Type;
                   Into_Set : in out Assembly_Set);

end Assembly_Set_Package;
```

The package body implements the set relationships as outlined earlier. For example, below is the implementation of the function "+" as it would appear in the package body:

```
-- Union

function "+" (Set_1, Set_2 : Assembly_Set)
             return Assembly_Set is
begin  -- "+"
    return Set_1 or Set_2;
end "+";
```

The code for the rest of the package body is not presented because the exercise asks the reader to provide a similar package body in its entirety.

Ideally, even greater separation of concerns between the abstract set and its Ada data structure is desirable so that a user cannot take advantage of the Boolean array implementation. The mechanism for enforcing this separation is beyond the scope of this exercise. It will be addressed in Exercise 3.2, on private types, and Exercise 6.1, on advanced set operations.

Another enhancement to this package could be to generalize it so that it can be easily adapted to other universes. Currently, a new package must be written for every new universe. Exercise 3.3, on generics, and Exercise 6.1 discuss ways of implementing general-purpose, reusable packages.

Problem

A common application of sets is in the realm of numbers. We frequently refer to the set of integers, the set of prime numbers, the set of natural numbers, and so forth.

Write a program that computes the following for numbers between 1 and 100 inclusive:

- the set of numbers divisible by 2, 3, or 5
- the set of numbers divisible by 2 or 3, but not by 5
- the set of numbers divisible by 3 and by 5
- the set of numbers not divisible by 3

Solution and Discussion

Let us explore the solution from a top-down point of view. The solution must compute the following sets:

{numbers between 1 and 100 divisible by 2, 3, or 5}

{numbers between 1 and 100 divisible by 2 or 3, but not by 5}

{numbers between 1 and 100 divisible both by 3 and by 5}

{numbers between 1 and 100 not divisible by 3}

Looking more closely at the requirements, we observe that there are 3 "building block" sets:

{numbers between 1 and 100 divisible by 2}

{numbers between 1 and 100 divisible by 3}

{numbers between 1 and 100 divisible by 5}

The four sets required for the solution are composed from the application of union, intersection, difference and complement on these three "basic" sets. The set of numbers divisible by 2, 3, or 5 is the union of the set of numbers divisible by 2, the set of numbers divisible by 3, and the set of numbers divisible by 5. Recall that union does not repeat elements in common more than once.

{numbers between 1 and 100 divisible by 2} +
{numbers between 1 and 100 divisible by 3} +
{numbers between 1 and 100 divisible by 5}

The set of numbers divisible by 2 or by 3 but not by 5, analogously, is computed by first taking the union of the set of numbers divisible by 2 and the set of numbers divisible by 3 and then eliminating all multiples of 5. Given the set of numbers divisible by 5, we simply take the difference between the previously computed union set and this set.

({numbers between 1 and 100 divisible by 2} +
{numbers between 1 and 100 divisible by 3}) -
{numbers between 1 and 100 divisible by 5}

The set of numbers divisible by both 3 and by 5 is the intersection of the set of numbers divisible by 3 and the set of numbers divisible by 5.

$\{$numbers between 1 and 100 divisible by 3$\}$ *
$\{$numbers between 1 and 100 divisible by 5$\}$

Lastly, the set of numbers not divisible by 3 is the complement of the set of numbers divisible by 3.

$\{$numbers between 1 and 100 divisible by 3$\}$ '

A set package for numbers between 1 and 100 inclusive will facilitate implementation of this solution. The specification of this package follows:

```
package Number_Set_Package is

    -- define elements of set

    subtype Number_Range is Integer range 1 .. 100;

    -- define set type

    type Number_Set is array (Number_Range) of Boolean;

    -- empty set:

    Empty_Set : constant Number_Set :=
                                 (Number_Range => False);

    -- Union

    function "+" (Set_1, Set_2 : Number_Set)
                 return Number_Set;

    -- Intersection

    function "*" (Set_1, Set_2 : Number_Set)
                 return Number_Set;

    -- Difference

    function "-" (Set_1, Set_2 : Number_Set)
                 return Number_Set;

    -- Complement

    function Complement (Set : Number_Set)
                        return Number_Set;
```

```
    -- Subset

    function Is_Subset (Set_1, Of_Set_2 : Number_Set)
                       return Boolean;

    -- Insertion

    procedure Insert (Element  : in Number_Range;
                      Into_Set : in out Number_Set);

end Number_Set_Package;
```

In the above specification, the universe set, i.e. the range of numbers, is given as a subtype of Integer so that the set of multiples of a factor outside this range may still be computed. For example, if the range were modified to include numbers from 10 to 110, the set of multiples of 3 could still be calculated.

An interesting alternative to explore is the use of renaming declarations. Because the operations of union, intersection and complement are implemented directly using Boolean operators, essentially renaming their logical counterparts, the package specification could have been written as follows (package entities which would be unchanged are miniaturized in the ellipses!):

```
package Number_Set_Package is

    .
    .
    .

    -- Union

    function "+" (Set_1, Set_2 : Number_Set)
                 return Number_Set renames "or";

    -- Intersection

    function "*" (Set_1, Set_2 : Number_Set)
                 return Number_Set renames "and";
```

```
-- Complement

function Complement (Set : Number_Set)
                    return Number_Set renames "not";

.
.
.

end Number_Set_Package;
```

This approach was not selected because it reveals the implementation of the set operations at the package specification level, contrary to the purpose of packaging the set as stated earlier in the Tutorial.

The problem is now reduced to one of computing the actual base sets. The expression

```
N mod Factor = 0
```

is true if and only if N is divisible by Factor. The first algorithm that comes to mind is quite simple and involves looping through all the numbers of the set, testing each one for *divisibility by Factor:*

```
Set := Number_Set_Package.Empty_Set;
for N in Set'Range
loop
    if N mod Factor = 0 then
        Insert (Element => N, Into_Set => Set);
    end if;
end loop;
```

This algorithm serves well for a small set of numbers; however it is inefficient over large sets of numbers. An equally clear algorithm consists of finding the multiplier which, when multiplied by Factor, yields the first multiple of Factor in the set, and then inserting elements into the set at every Multiplier + Factor'th position. The first part of this algorithm is accomplished by the initialization of the local variable Set, and the second part is performed by the for loop in the function body. The attributes First and Last are used to tie any dependence on the range of the set directly to the bounds listed in its declaration, thereby facilitating maintenance.

```
function Multiples_of (Factor : Integer)
                       return Number_Set is

     -- Assume initially no numbers in set divisible by Factor

     Set  : Number_Set := Empty_Set;

     --  The subtype All_Divisors has a range from the first
     --  multiple of Factor in the set to the last multiple
     --  of Factor in the set

     subtype All_Divisors is Natural range
              (Number_Range'First + Factor - 1) / Factor ..
               Number_Range'Last/Factor;
begin  -- Multiples_of

     for N in All_Divisors
     loop
          Insert (Element => N * Factor, Into_Set => Set);
     end loop;

     return Set;

end Multiples_of;
```

All that is left is the main procedure to tie these fragments together. It contains the object declarations for the three basic sets and for the four result sets:

```
Multiples_2 : constant Number_Set := Multiples_of (2);
Multiples_3 : constant Number_Set := Multiples_of (3);
Multiples_5 : constant Number_Set := Multiples_of (5);
Multiples_2_3_5,
Multiples_2_3_not_5,
Multiples_3_and_5,
Not_Multiples_3     : Number_Set;
```

Notice how the objects for the three base sets are declared to be constants. These sets are fundamental to all the otner set manipulations and they will not vary for the duration of the problem. Furthermore, by putting them in the declarative portion of the main procedure, they

unclutter the computation of the four required sets in the executable portion because they are not there to "distract" the reader. The numbers between 1 and 100 divisible by 2 are established through the function call to Multiples_of in the initialization of the constant. This technique presupposes that the function Multiples_of is a separately compiled library unit which may be accessed by the main procedure. The elements of the other two basic sets are similarly established. The result sets are computed by applying the set operations provided in Number_Set_Package. For example, the set of numbers divisible by both 3 and 5 is

```
Multiples_3_and_5 := Multiples_3 * Multiples_5;
```

This set could have been computed just as easily by writing:

```
Multiples_3_and_5 := Multiples_of (3) *
                          Multiples_of (5);
```

however, the purpose of the exercise is to manipulate the sets of numbers divisible by 2, 3, and 5 using the set relationships union, intersection, difference, complement and subset.

The complete solution code follows:

```
package Number_Set_Package is

    -- define elements of set

    subtype Number_Range is Integer range 1 .. 100;

    -- define set type

    type Number_Set is array (Number_Range) of Boolean;

    -- empty set:

    Empty_Set : constant Number_Set :=
                              (Number_Range => False);

    -- Union

    function "+" (Set_1, Set_2 : Number_Set)
                 return Number_Set;
```

```
    -- Intersection

    function "*" (Set_1, Set_2 : Number_Set)
                 return Number_Set;

    -- Difference

    function "-" (Set_1, Set_2 : Number_Set)
                 return Number_Set;

    -- Complement

    function Complement (Set : Number_Set)
                        return Number_Set;
    -- Subset

    function Is_Subset (Set_1, Of_Set_2 : Number_Set)
                       return Boolean;

    -- Insertion

    procedure Insert (Element  : in Number_Range;
                      Into_Set : in out Number_Set);

end Number_Set_Package;


package body Number_Set_Package is

    -- Union

    function "+" (Set_1, Set_2 : Number_Set)
                 return Number_Set is
    begin  -- "+"
        return Set_1 or Set_2;
    end "+";

    -- Intersection

    function "*" (Set_1, Set_2 : Number_Set)
                 return Number_Set is
    begin  -- "*"
        return Set_1 and Set_2;
    end "*";

    -- Difference

    function "-" (Set_1, Set_2 : Number_Set)
                 return Number_Set is
    begin  -- "-"
        return Set_1 and not Set_2;
    end "-";
```

```
    -- Complement

    function Complement (Set : Number_Set)
                        return Number_Set is
    begin  -- Complement
        return not Set;
    end Complement;

    -- Subset

    function Is_Subset (Set_1, Of_Set_2 : Number_Set)
                       return Boolean is
    begin  -- Is_Subset
        return (Set_1 and Of_Set_2) = Set_1;
    end Is_Subset;

    -- Insertion

    procedure Insert (Element  : in Number_Range;
                      Into_Set : in out Number_Set) is
    begin  -- Insert
        Into_Set (Element) := True;
    end Insert;

end Number_Set_Package;

with Number_Set_Package; use Number_Set_Package;
function Multiples_of (Factor : Integer)
                      return Number_Set is

    -- Assume initially no numbers in set divisible by Factor

    Set  : Number_Set := Empty_Set;

    -- The subtype All_Divisors has a range from the first
    -- multiple of Factor in the set to the last multiple
    -- of Factor in the set.

   subtype All_Divisors is Natural range
          (Number_Range'First + Factor - 1) / Factor ..
           Number_Range'Last/Factor;
```

```
begin  -- Multiples_of

    for N in All_Divisors
    loop
        Insert (Element => N * Factor, Into_Set => Set);
    end loop;

    return Set;

end Multiples_of;

with Number_Set_Package; use Number_Set_Package;
with Multiples_of;
procedure Compute_Multiples is

    Multiples_2 : constant Number_Set := Multiples_of (2);
    Multiples_3 : constant Number_Set := Multiples_of (3);
    Multiples_5 : constant Number_Set := Multiples_of (5);
    Multiples_2_3_5,
    Multiples_2_3_not_5,
    Multiples_3_and_5,
    Not_Multiples_3     : Number_Set;

begin  -- Compute_Multiples

    -- Compute multiples of 2, 3, or 5

    Multiples_2_3_5 := Multiples_2 + Multiples_3 + Multiples_5;

    -- Compute multiples of 2 or 3, but not 5

    Multiples_2_3_not_5 := (Multiples_2 + Multiples_3) -
                                       Multiples_5;

    -- Compute multiples of 3 and 5

    Multiples_3_and_5 := Multiples_3 * Multiples_5;

    -- Compute numbers not divisible by 3

    Not_Multiples_3 := Complement (Multiples_3);

end Compute_Multiples;
```

Several design decisions were made in writing the above solution. The three basic set variables are not strictly necessary; repeated calls to the divisibility function would have had the same effect:

```
Multiples_3_and_5 := Multiples_of (3) * Multiples_of (5);
```

For efficiency reasons, this approach was not chosen. These basic sets are used several times, and it makes sense to compute them once and save the result. If one wanted to examine the contents of these sets, it is useful to be able to access the appropriate object.

Another design decision was to abstract the computation of the multiples sets into a function. The divisibility algorithm was extensively applied throughout the problem solution; writing it at every point of usage would have cluttered the code and obscured the functionality of the main program. By having a distinct and therefore possibly reusable function, the software clarity and maintainability is enhanced. Furthermore, this function was made into a library unit so that it could be used in the constant declarations for Multiples_2, Multiples_3, and Multiples_5. It is separately compiled and because it appears in a context clause preceding the main procedure Compute_Multiples, it will be elaborated (and therefore usable) before being called in the constant declarations.

The reasons behind packaging the set operations were explained earlier. Here again, failure to do so would have a negative impact on program readability and maintainability. Moreover, it would eclipse the simplicity of the set data abstraction.

EXERCISE 2.2

ACCESS TYPES: STRUCTURE, USE, AND ALLOCATORS

Objective

This exercise introduces access types and the language rules governing their use, and additionally examines circumstances in which they occur.

Tutorial

An object of an access type is a reference or pointer to another object and may be likened in certain respects to an address. For example, in the illustration of memory space below,

the left box is of an access type. It is said to point to, or designate, the value in the right box. If the declared type of the object stored in the right hand box is used, for example, in a program to drive a plotter,

```
type Nib_Type is (Fine, Medium, Large);
```

for the kind of pen nib on a plotter, the type declaration for the access to it would be written,

```
type Nib_Pointer_Type is access Nib_Type;
```

In an access type declaration, the type of the variables which may be referenced by an object of the access type (e.g. Nib_Type in the declaration directly above) is included, and indeed it must be. An access value may not point to any object, but only to an object of the type specified in the access type declaration. One could not write,

```
type A_Type is access;              -- **ILLEGAL
```

because there is no type specified for the allocated variables.

An access type declaration takes the form,

```
type Access_Type_Name is
     access Designated_Subtype_Name [Constraint];
```

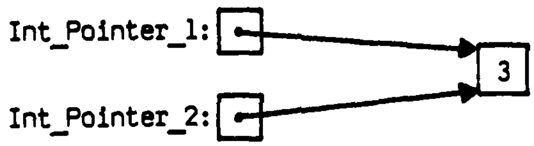
where the designated subtype name and optional constraint specify the subtype of the variables that are pointed to by the access type. (The subtype of the objects pointed to is called the designated subtype.) There are not any particular limitations on the designated type of an access type. The declaration,

```
type Integer_Pointer_Type is access Integer range 0 .. 500;
```

defines an access type for objects that point to integer values in the range 0 to 500. One could then create two objects of that access type by writing,

```
Int_Pointer_1, Int_Pointer_2 : Integer_Pointer_Type;
```

Int_Pointer_1 and Int_Pointer_2 are declared access objects which contain access values. This declaration is equivalent, in every respect, to two separate declarations for Int_Pointer_1 and Int_Pointer_2. They both reference the same type of allocated variables, and, because they are of the same type, may point to the same space in memory, as depicted in the illustration below,



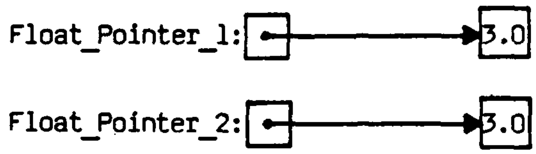
where the right hand box contains a value of type Integer range 0 .. 500.

The box containing the value 3 in the illustration directly above is an allocated variable, which is entirely different from a declared variable. It arises from dynamic allocation (the creation of an access object during program execution). Allocated variables and declared variables may not mix; an access type may never point to a declared variable, and an allocated variable may be referred to only by an access value.

Naturally, one may declare many types (and objects of those types) having the same type of designated variable, as in

```
type Float_Pointer_Type_1 is access Float;
type Float_Pointer_Type_2 is access Float;
Float_Pointer_1 : Float_Pointer_Type_1;
Float_Pointer_2 : Float_Pointer_Type_2;
```

But in this case, Float_Pointer_1 and Float_Pointer_2 may not designate the same allocated variable, even though they point to variables of the same type. The corresponding illustration of memory would be:



where both of the right hand boxes are of type Float. As the diagram suggests, the collection of allocated variables associated with an object that is declared of an access type is not shared with (cannot be accessed by) an object of any other access type.

An allocator takes one of several forms:

```
new Type_or_Subtype_Name ['(Initial Value)]
new Unconstrained_Array_Subtype_Name Index_Constraint
new Unconstrained_Record_Subtype_Name
                          Discriminant_Constraint
```

The first two forms will be discussed in this tutorial. The last cannot be explained until the reader has been introduced to discriminants, which are addressed in Exercise 3.1 of this book.

Using the type declaration for a pointer to a Nib_Type, as mentioned earlier, we can declare an object, and provide an allocator, written

```
Nib_Pointer_1 : Nib_Pointer_Type := new Nib_Type;
```

Here the allocated variable takes the first and simplest form listed above. There is no value given. This statement creates an allocated variable of type Nib_Type, presumably to be assigned a value later in the program, and assigns an access value designating that variable to Nib_Pointer_1.

Using the optional constraint in the first form for an allocator listed above, one could give an initial value to the Nib_Type allocated variable by writing,

```
Nib_Pointer_1 : Nib_Pointer_Type := new Nib_Type'(Fine);
```

where Fine is the initial value.

An object of an access type need not be initialized. One can simply write:

```
Nib_Pointer_2 : Nib_Pointer_Type;
```

Because no initial value is listed, a null pointer is assigned by default. The reserved word null names a pointer to null, specifically indicating that the pointer does not point to a designated variable. A null pointer may be denoted, as in,

```
Nib_Pointer_3 : Nib_Pointer_Type := null;
```

which has the same effect as the omission of an initial value.

When an allocator names an unconstrained array type, the second form for an allocator in the list above must be used. Either an index constraint or an initial value is given. For example, if we create an array of the names of all the chaplains in the French Foreign Legion,

```
subtype Chaplain_Name_Type is String (1 .. 15);
type Chaplains_in_FFL_Type is
        array (Positive range <>) of Chaplain_Name_Type;
```

and an access pointing to it,

```
type Chaplains_in_FFL_Ptr_Type is
        access Chaplains_in_FFL_Type;
Chaplains_1, Chaplains_2, Chaplains_3 :
        Chaplains_in_FFL_Ptr_Type;
```

values may be assigned to these objects using either of the two forms below:

```
Chaplains_1 := new Chaplains_in_FFL_Type (1 .. 1250);
Chaplains_2 := new Chaplains_in_FFL_Type'("Anna Pavlova   ",
                                          "Marvin Gaye    ",
                                          "Max Robespierre",
                                          "A.J.P. Taylor  ",
                                          "Edgar Allen Poe");
```

In the first assignment an index constraint is specified; in the second, an array aggregate is given for the initial value of the array. Notice that there is an apostrophe in the expression of the allocator having an initial value,

but not for the index constraint. Also, in the initialization, it is not necessary to include two sets of parentheses. To repeat, because Chaplains_in_FFL_Type is an unconstrained array, one could not simply write,

```
Chaplains_3 := new Chaplains_in_FFL_Type;  -- **ILLEGAL
```

In order to refer to the value of an entire allocated variable (a process that is often called dereferencing), one uses ".all" . For instance, to obtain the value pointed to by Nib_Pointer_1, we write,

```
Nib_Pointer_1.all
```

which evaluates to the enumeration literal Fine. Similarly, the array pointed to by Chaplains_2 above could be referred to by writing,

```
Chaplains_2.all
```

This expression may be used on either side of an assignment. It delivers the entire allocated variable, in this case, the entire array. If we want to reference only one or several components of the array pointed to by Chaplains_2 an index should be given. For the second component we could write,

```
Chaplains_2.all(2)
```

but the ".all" is unnecessary.

There are four different types of notation for referring to allocated variables. They are for (1) an entire allocated variable, (2) a component of an allocated array, (3) a slice of an allocated array, and (4) a component of an allocated record.

A component of an allocated array may be referenced by writing just the name of the access object designating that array, followed by a parenthetical index. For example, to return to the Chaplains, the third component of the array to which it points is referenced by writing,

```
Chaplains_2(3)
```

where Chaplains_2 is a pointer to the array. To change the value of the third component of the allocated array from "Max Robespierre" to "Giles Nicholson", one could write,

```
Chaplains_2(3) := "Giles Nicholson";
```

And to assign the value of the third component of the array to some variable,

```
First_Chaplain : Chaplain_Name_Type;
```

one would simply write,

```
First_Chaplain := Chaplains_2(3);
```

We can also declare an access type for the names of the chaplains in the French Foreign Legion, rather than, as in the example above, an access to an array of their names:

```
type Chaplain_Name_Ptr_Type is access Chaplain_Name_Type;
```

If "Reza Pahlavi   " is given as the value of one such string, as follows,

```
Chaplain_Name_Ptr : Chaplain_Name_Ptr_Type :=
        new String'("Reza Pahlavi   ");
```

a slice of the allocated string may be referenced simply by writing the name of the pointer to the string, Chaplain_Name_Ptr, followed by parenthetical indices. For example, the first name in the declaration above could be changed as follows,

```
Chaplain_Name_Ptr (1 .. 4) := String'("Shah");
```

leaving the name "Shah Pahlavi   " in the allocated string variable.

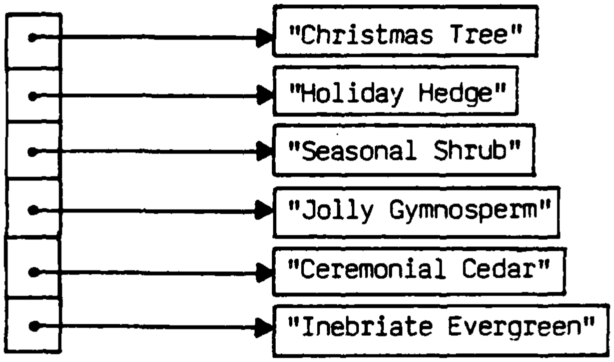
As you may have noticed, in the array of the names of the chaplains above, some of the names were padded with blanks because the length of each component was declared to be fifteen characters. Each chaplain's name must have the same length because String is an unconstrained array type, and in arrays of arrays the component subtype must be constrained, with a constraint that applies to all components.

Access types can be used to circumvent this problem, that is, by defining "ragged" arrays, which are arrays whose components are not necessarily of the same length. For example, because of separation of Church and DoD, alternative names for the Pentagon's Christmas Tree must be determined and stored in an array. These names may be of different lengths if we write,

```
type Name_Pointer_Type is access String;
type Xmas_Tree_Name_Type is
     array (Positive range <>) of Name_Pointer_Type;
Xmas_Tree_Name_List : Xmas_Tree_Name_Type (1 .. 6) :=
                  (new String'("Christmas Tree"),
                   new String'("Holiday Hedge"),
                   new String'("Seasonal Shrub"),
                   new String'("Jolly Gymnosperm"),
                   new String'("Ceremonial Cedar"),
                   new String'("Inebriate Evergreen"));
```

So, we have created an array of pointers to strings. In the declaration of Xmas_Tree_Name_List an index constraint must be given, or the object must be declared constant (in which case the constraint on the array object Xmas_Tree_Name_Pointer_List would be determined from the initial value). In memory, the structure would look like:

Xmas_Tree_Name_List:



Each allocated string is constrained, so we cannot write,

```
Xmas_Tree_Name_List(1).all := "Festive Fir";
```

Instead, we specifically create an allocated string, as in,

```
Xmas_Tree_Name_List(1).all := new String'("Festive Fir");
```

In order to enable easy expansion of this list of synonyms, an access type to Xmas_Tree_Name_Type would have to be created:

```
type Xmas_Tree_Name_Ptr_Type is access Xmas_Tree_Name_Type;
```

Then, to add another synonym, we can allocate a new, larger array, whose values are formed by the catenation of the original name list and the seventh synonym:

```
Xmas_Tree_Name_List_Ptr_1 : Xmas_Tree_Name_Ptr_Type :=
        new Xmas_Tree_Name_Type'
                (new String'("Christmas Tree"),
                 new String'("Holiday Hedge"),
                 new String'("Seasonal Shrub"),
                 new String'("Jolly Gymnosperm"),
                 new String'("Ceremonial Cedar"),
                 new String'("Inebriate Evergreen"));

Xmas_Tree_Name_List_Ptr_2 : Xmas_Tree_Name_Ptr_Type :=
        new Xmas_Tree_Name_Type'
                (Xmas_Tree_Name_List_Ptr_1.all &
                        new String'("Convivial Conifer"));
```

We could then assign the value of Ptr_2 to Ptr_1, as follows,

```
Xmas_Tree_Name_List_Ptr_1 := Xmas_Tree_Name_List_Ptr_2;
```

which would leave a pointer to the expanded allocated array in the original access object, Xmas_Tree_Name_List_Ptr_1. Another synonym could then be added by catenating that array with a pointer to the new phrase. So, the role of Ptr_1 is to store the current array, and the role of Ptr_2 is to temporarily store the expanded array. The values of the corresponding allocated variables are easily exchanged by assigning one access value to the other.

Like the components of allocated arrays, components of allocated records are referenced simply by writing the name of the access object, followed by the component name. For example, given a record type declaration for a cabinet post and an access object that points to it,

```
type Department_Type is (State, Treasury, Defense, Interior,
                         Agriculture, Commerce, Labor, Education,
                         Energy, Housing_and_Urban_Development,
                         Health_and_Human_Services, Transportation);
type Cabinet_Post_Type is
    record
        Department      : Department_Type;
        Cabinet_Officer : String (1 .. 15);
        Annual_Budget   : Integer;
    end record;

type Cabinet_Post_Ptr_Type is access Cabinet_Post_Type;
```

one can declare an object of Cabinet_Post_Ptr_Type, initialized to the Department of the Treasury,

```
Cabinet_Post_Ptr : Cabinet_Post_Ptr_Type :=
        new Cabinet_Post_Type'
                (Department         => Treasury,
                 Department_Officer => "Elmer Peters   ",
                 Annual_Budget      => 4E9);
```

A component of the allocated record, for example the name of the department officer, can be referenced by writing,

```
Cabinet_Post_Ptr.Department_Officer
```

which evaluates to "Elmer Peters   ". To change a component of the array one could write,

```
Cabinet_Post_Ptr.Department_Officer := "Joel Goethe    ";
```

And, as mentioned,

```
Cabinet_Post_Ptr.all
```

delivers the entire record.

To conclude, access values are used primarily (1) to dynamically change the role of certain variables, (2) to provide access to shared data, (3) to avoid moving large amounts of data, (4) to build arrays that have components of varying lengths, and (5) to build recursive data structures, which are the subject of the next exercise.

Problem

The problems in Exercises 1.1 and 1.2 involved the conversion of foreign units of measure to the English system (familiarity with that problem is necessay for the completion of this one). Change the package written in the solution of Exercise 1.2, English_Measurements, so that the length of the table of conversion values is unconstrained, and most easily expanded during program execution.

Discussion and Solution

An object of the array of records used to store the conversion information, of Conversion_Table_Type, cannot be declared without placing a constraint upon the length of the array. However, if we define the conversion table as an access table designating an array type, arrays of various sizes may be allocated. (A given allocated array is bounded.)

In the solution for Exercise 1.2 the declarations for the conversion table appear:

```
type Foreign_Unit_Type is
    record
        Unit_Name         : Foreign_Unit_Name_Type;
        Conversion_Unit   : English_Units_Type;
        Conversion_Factor : Positive_Real;
    end record;

type Conversion_Table_Type is array (Natural range <>)
        of Foreign_Unit_Type;
```

These definitions may remain exactly as written. We add only the access to an array of Conversion_Table_Type:

```
type Conversion_Table_Ptr_Type is access Conversion_Table_Type;
```

The problem is now essentially solved. We must only examine the effect of this new type declaration upon the rest of the code, and then rewrite it accordingly.

The specification of English_Measurements does not change, nor do any of the declarations in the declarative region of its body, aside from the addition of the access type above and the creation of the actual conversion table (the object). The table is defined in the internal package Measure_Table, which is declared and "use"ed inside English_Measurements. That package appears in Exercise 1.2:

```
package Measure_Table is
    Size : Natural := Table_Size;
    Conversion_Table : Conversion_Table_Type (1 .. Size + 10);
end Measure_Table;
```

where 10 is the maximum number of additions that can be made to the table in one run. Rather than a table, we want an access object that points to a table, which is written,

```
Conversion_Table_Ptr : Conversion_Table_Ptr_Type :=
                       new Conversion_Table_Type (1 .. Size);
```

The initialization is not required here, but at some point in the program the space for the allocated table must be created.

In the solution of Exercise 1.2 the statements in English_Measurements, which read the conversion information into the table, are as follows:

```
Measure_IO.Open (Table_File, Measure_IO.In_File, Measure_Table_Name);
for Index in 1 .. Size
loop
   Measure_IO.Read (Table_File,
                    Conversion_Table(Index),
                    Measure_IO.Positive_Count (Index));
end loop;
Measure_IO.Close (Table_File);
```

The only change that need be made is to the statement in the loop, where Conversion_Table, instead of Conversion_Table_Ptr, is given as an out parameter. The new solution is simply,

```
Measure_IO.Open (Table_File, Measure_IO.In_File, Measure_Table_Name);
for Index in 1 .. Size
loop
   Measure_IO.Read (Table_File,
                    Conversion_Table_Ptr(Index),
                    Measure_IO.Positive_Count (Index));
end loop;
Measure_IO.Close (Table_File);
```

As mentioned, the specifications of the subprograms in English_Measurements are unaffected by the introduction of the access to the table. But the bodies of those that alter, use, or even refer to the table, will require some rewriting. Happily, only the procedures Add_Conversion_To_Table and Look_Up_Conversion_Values are dependent on the structure of the conversion table, so we examine their code from Chapter One.

In the solution of Exercise 1.2 Add_Conversion_To_Table is stubbed out and separately written:

```
separate (English_Measurements)
procedure Add_Conversion_To_Table
    (Name         : in Foreign_Unit_Name_Type;
     English_Unit : in English_Units_Type;
     Factor       : in Positive_Real) is

    New_Measure : Foreign_Unit_Type :=
                    (Unit_Name         => Name,
                     Conversion_Unit   => English_Unit,
                     Conversion_Factor => Factor);

begin -- Add_Conversion_To_Table

    -- if room in table

    if Size < Conversion_Table'Last then

        --  Update External Table

        Size := Size + 1;
        Measure_IO.Open  (Table_File,
                          Measure_IO.Inout_File,
                          Measure_Table_Name);
        Measure_IO.Write (Table_File, New_Measure, Size);
        Measure_IO.Close (Table_File);

        --  Update Internal File

        Conversion_Table (Size) := New_Measure;

    else

        raise Table_Full;

    end if;

end Add_Conversion_To_Table;
```

This procedure must be fundamentally changed because it is specifically in the expansion of the table that the use of the access type enhances the serviceabilility of the package. The best algorithm for Add_Conversion_To_Table is to (1) increment Size, the current length of the table, (2) add the new record to the external file, (3) create a new access to a new allocated table, defined to be longer than the first table by one, (4) copy the old allocated table into the new allocated table, (5) put the new record into the last component of the new allocated table, and (6) assign to the original table pointer the value of

the new one. Because it is no longer possible to exceed the size of the table, it is not necessary to check if it is full, nor, of course, to raise the exception Table_Full (whose declaration in the body of English_Measurements should also be removed). So the procedure is written,

```
separate (English_Measurements)
procedure Add_Conversion_To_Table
        (Name         : in Foreign_Unit_Name_Type;
         English_Unit : in English_Units_Type;
         Factor       : in Positive_Real) is

    New_Measure   : Foreign_Unit_Type :=
                   (Unit_Name         => Name,
                    Conversion_Unit   => English_Unit,
                    Conversion_Factor => Factor);

    New_Table_Ptr : Conversion_Table_Ptr_Type :=
                    new Conversion_Table_Type (1 .. Size + 1);

begin -- Add_Conversion_To_Table

    Size := Size + 1;

    -- Update external file.

    Measure_IO.Open (Table_File,
                     Measure_IO.Inout_File,
                     Measure_Table_Name);

    Measure_IO.Write (Table_File,
                      New_Measure,
                      Measure_IO.Positive_Count (Size));

    Measure_IO.Close (Table_File);

    -- Update internal file.

    New_Table_Ptr (1 .. Size - 1) := Conversion_Table_Ptr.all;
    New_Table_Ptr (Size) := New_Measure;
    Conversion_Table_Ptr := New_Table_Ptr;

end Add_Conversion_To_Table;
```

The last section of this procedure illustrates one of the most valuable uses of access types, that is, the ability to easily switch the role of a variable (the new allocated table becomes the regular allocated table). Also, this conversion table could conceivably become quite large, which would make the

assignment of a mere access value significantly more attractive than that of a lengthy array of records.

The procedure Look_Up_Conversion_Values is given in the solution of Exercise 1.1, and is written:

```
procedure Look_Up_Conversion_Values
        (Foreign_Unit_Name : in Foreign_Unit_Name_Type;
         Conversion_Unit   : out English_Units_Type;
         Conversion_Factor : out Positive_Real) is

begin       -- Look_Up_Conversion

    Conversion_Factor := 0.0;

    for I in Conversion_Table'Range
    loop
        if Conversion_Table(I).Unit_Name = Foreign_Unit_Name then
            Conversion_Unit   := Conversion_Table(I).Conversion_Unit;
            Conversion_Factor := Conversion_Table(I).Conversion_Factor;
        end if;
    end loop;

    if Conversion_Factor = 0.0 then
        raise Measure_Not_Found;
    end if;

end Look_Up_Conversion_Values;
```

The only changes to be made are the references to Conversion_Table in the loop statement. As you recall, the expressions for a specific component in an array, and for an access to a component in an allocated array have the same form. So we simply replace Conversion_Table with Conversion_Table_Ptr, as follows:

```
    for I in Conversion_Table_Ptr'Range
    loop
        if Conversion_Table_Ptr(I).Unit_Name = Foreign_Unit_Name then
            Conversion_Unit   :=
                    Conversion_Table_Ptr(I).Conversion_Unit;
            Conversion_Factor :=
                    Conversion_Table_Ptr(I).Conversion_Factor;
        end if;
    end loop;
```

The rest of the procedure remains the same.

English_Measurements appears in its entirety:

```
package English_Measurements is

    type Positive_Real is digits 5 range 0.0 .. 5000.0;

    type English_Units_Type is (Inches, Feet, Yards, Miles);

    type Measurement_Type is
        record
            Inches : Positive_Real range 0.0 .. 12.0;
            Feet   : Positive_Real range 0.0 .. 3.0;
            Yards  : Positive_Real range 0.0 .. 1760.0;
            Miles  : Positive_Real;
        end record;

    subtype Foreign_Unit_Name_Type is String (1 .. 10);

    Measure_Not_Found : exception;

    function Conversion
            (Foreign_Unit_Name : Foreign_Unit_Name_Type;
             Number_of_Units   : Positive_Real)
            return Measurement_Type;

    procedure Add_Conversion_To_Table
            (Name         : in Foreign_Unit_Name_Type;
             English_Unit : in English_Units_Type;
             Factor       : in Positive_Real);

end English_Measurements;

with Direct_IO;
package body English_Measurements is

    type Foreign_Unit_Type is
        record
            Unit_Name          : Foreign_Unit_Name_Type;
            Conversion_Unit    : English_Units_Type;
            Conversion_Factor  : Positive_Real;
        end record;

    type Conversion_Table_Type is array (Natural range <>)
            of Foreign_Unit_Type;

    type Conversion_Table_Ptr_Type is access Conversion_Table_Type;
```

```
package Measure_IO is new Direct_IO (Foreign_Unit_Type);

Table_File         : Measure_IO.File_Type;
Measure_Table_Name : constant String := "Measurement_Table.Dat";

function Table_Size return Natural is

    Size : Natural;

begin -- Table_Size

    Measure_IO.Open (Table_File,
                     Measure_IO.In_File,
                     Measure_Table_Name);
    Size := Natural (Measure_IO.Size (Table_File));
    Measure_IO.Close (Table_File);
    return Size;

end Table_Size;

package Measure_Table is

    Size : Natural := Table_Size;
    Conversion_Table_Ptr : Conversion_Table_Ptr_Type :=
                           new Conversion_Table_Type (1 .. Size);

end Measure_Table;

use Measure_Table;

function Conversion
        (Foreign_Unit_Name : Foreign_Unit_Name_Type;
         Number_of_Units   : Positive_Real)
        return Measurement_Type is separate;

procedure Look_Up_Conversion_Values
        (Foreign_Unit_Name : in  Foreign_Unit_Name_Type;
         Conversion_Unit   : out English_Units_Type;
         Conversion_Factor : out Positive_Real)
        is separate;

function Adjusted_Measurements
        (Quantity : Positive_Real;
         Unit     : English_Units_Type)
        return Measurement_Type is separate;

procedure Add_Conversion_To_Table
        (Name         : in Foreign_Unit_Name_Type;
         English_Unit : in English_Units_Type;
         Factor       : in Positive_Real)
        is separate;
```

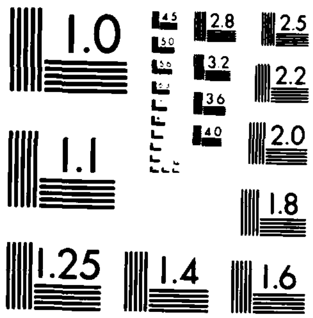
```
begin -- English_Measurements

    Measure_IO.Open (Table_File,
                     Measure_IO.In_File,
                     Measure_Table_Name);
    for Index in 1 .. Size
    loop
        Measure_IO.Read (Table_File,
                         Conversion_Table_Ptr(Index),
                         Measure_IO.Positive_Count (Index));
    end loop;
    Measure_IO.Close (Table_File);

end English_Measurements;

separate (English_Measurements)
function Adjusted_Measurements
        (Quantity : Positive_Real;
         Unit     : English_Units_Type)
         return Measurement_Type is

    type Intermediate_Type is
            array (English_Units_Type) of Positive_Real;

    Msmnt  : Intermediate_Type;
    Result : Measurement_Type;

    function Decimal_Part (X : Positive_Real) return Positive_Real is
    begin      -- Decimal_Part
        return X - Positive_Real (Integer (X - 0.5));
    end Decimal_Part;

begin      -- Adjusted_Measurements

    Msmnt(Unit) := Quantity;
    if Decimal_Part (Msmnt(Miles)) > 0.0 then
        Msmnt(Y rds) := Msmnt(Yards) +
          Decimal_Part (Msmnt(Miles)) * 1760.0;
    end if;

    if Decimal_Part (Msmnt(Yards)) > 0.0 then
        Msmnt(Feet) := Msmnt(Feet) +
                Decimal_Part (Msmnt(Yards)) * 3.0;
    end if;

    if Decimal_Part (Msmnt(Feet)) > 0.0 then
        Msmnt(Inches) := Msmnt(Inches) +
                Decimal_Part (Msmnt(Feet)) * 12.0;
    end if;
```

```
    if Msmnt(Inches) >= 12.0 then
        Msmnt(Feet) := Msmnt(Feet) +
            Positive_Real (Integer (Msmnt(Inches) - 0.5) / 12);
        Msmnt(Inches) := Msmnt(Inches) -
            Positive_Real (Integer (Msmnt(Inches) - 0.5) / 12) * 12.0;
    end if;

    if Msmnt(Feet) >= 3.0 then
        Msmnt(Yards) := Msmnt(Yards) +
                Positive_Real (Integer (Msmnt(Feet) - 0.5) / 3);
        Msmnt(Feet) :=
                Positive_Real (Integer (Msmnt(Feet) - 0.5) mod 3);
    end if;

    if Msmnt(Yards) >= 1760.0 then
        Msmnt(Miles) := Msmnt(Miles) +
                Positive_Real (Integer (Msmnt(Yards) - 0.5) / 1760);
        Msmnt(Yards) :=
                Positive_Real (Integer (Msmnt(Yards) - 0.5) mod 1760);
    end if;

    Result.Inches := Msmnt(Inches);
    Result.Feet   := Msmnt(Feet);
    Result.Yards  := Msmnt(Yards);
    Result.Miles  := Msmnt(Miles);

    return Result;

end Adjusted_Measurements;


separate (English_Measurements)
procedure Look_Up_Conversion_Values
        (Foreign_Unit_Name : in Foreign_Unit_Name_Type;
         Conversion_Unit   : out English_Units_Type;
         Conversion_Factor : out Positive_Real) is

begin       -- Look_Up_Conversion

    Conversion_Factor := 0.0;

    for I in Conversion_Table_Ptr'Range
    loop
        if Conversion_Table_Ptr(I).Unit_Name = Foreign_Unit_Name then
            Conversion_Unit   :=
                    Conversion_Table_Ptr(I).Conversion_Unit;
            Conversion_Factor :=
                    Conversion_Table_Ptr(I).Conversion_Factor;
        end if;
    end loop;
```

```
    if Conversion_Factor = 0.0 then
       raise Measure_Not_Found;
    end if;

end Look_Up_Conversion_Values;


separate (English_Measurements)
function Conversion
       (Foreign_Unit_Name : Foreign_Unit_Name_Type;
        Number_of_Units   : Positive_Real)
        return Measurement_Type is

    Conversion_Factor : Positive_Real;
    Conversion_Unit   : English_Units_Type;

begin

    Look_Up_Conversion_Values
          (Foreign_Unit_Name, Conversion_Factor, Conversion_Unit);

    return Adjusted_Measurements
          (Number_of_Units * Conversion_Factor, Conversion_Unit);

end Conversion;

separate (English_Measurements)
procedure Add_Conversion_To_Table
        (Name         : in Foreign_Unit_Name_Type;
         English_Unit : in English_Units_Type;
         Factor       : in Positive_Real) is

    New_Measure   : Foreign_Unit_Type :=
                   (Unit_Name          => Name,
                    Conversion_Unit    => English_Unit,
                    Conversion_Factor  => Factor);

    New_Table_Ptr : Conversion_Table_Ptr_Type :=
                    new Conversion_Table_Type (1 .. Size + 1);

begin -- Add_Conversion_To_Table

    Size := Size + 1;

    -- Update external file.

    Measure_IO.Open (Table_File,
                     Measure_IO.Inout_File,
                     Measure_Table_Name);
```

AD-A146 257 ADVANCED ADA (TRADEMARK)(U) SOFTECH INC WALTHAM MA 2/5
JUL 84 DAAB07-83-C-K514

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

```
        Measure_IO.Write (Table_File,
                          New_Measure,
                          Measure_IO.Positive_Count (Size));

        Measure_IO.Close (Table_File);

        -- Update internal file.

        New_Table_Ptr (1 .. Size - 1) := Conversion_Table_Ptr.all;
        New_Table_Ptr (Size)          := New_Measure;
        Conversion_Table_Ptr          := New_Table_Ptr;

    end Add_Conversion_To_Table;
```

It is worth noting how easily the modification of a central data structure has been incorporated into English_Measurements. Aside from reworking Add_Conversion_To_Table, alterations have primarily consisted of a global change from Conversion_Table to Conversion_Table_Ptr. Even this could have been avoided by keeping the same name, and only changing its type. The fact that the form for referencing a component of an allocated array is the same as that of a component in an ordinary array has greatly facilitated the incorporation of the access type. But more importantly, the problem has been quite simple to solve because English_Measurements was already well designed.

Indeed, the ability to easily change the structures or specifications of a program is a principal characteristic of good programming design. English_Measurements was written using fairly strict data encapsulation, that is, each parcel of data and each action is separate, and has a clear interface with the rest of the program. There are not many global variables, and few assumptions are made about the structure of those that there are. The subprograms in English_Measurements are adaptable.

EXERCISE 2.3

RECURSIVE TYPES

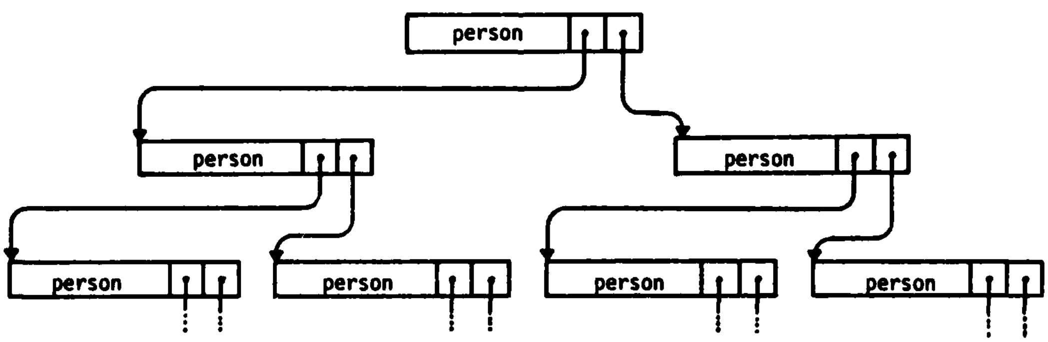
Objective

This tutorial introduces the concepts of recursive data structures using examples of trees, linked lists, stacks, and queues, and then examines the Ada implementation of these structures and of the basic operations on them.

Tutorial

A recursive type is used to represent a structure which has a component with the same definition. For example, the data structure used to represent a person would contain information about the person's parents. Each parent, in turn, would be a person. This is the essence of a recursive data structure. A recursive type is usually implemented as a record, commonly called a node, with two kinds of components, data components and access components.

The data components of the node contain all the information about some item, and the access components "point to" other nodes. For example, in the structure for a person the data components contain information such as Age, Name, Birthdate, Social_Security_Number, etc., and the access components contain pointers to the nodes of the person's parents. Pictorially this structure looks like:

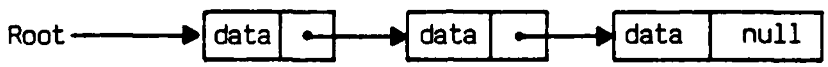


This particular recursive structure illustrates a tree structure. It has a root, the original Person, and descendants, the other nodes "pointed to." At the bottom of the tree, the nodes do not point anywhere further, so their

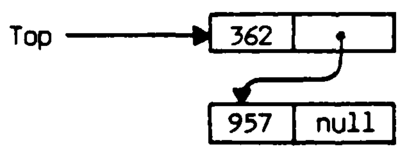
pointers are explicitly "grounded." A tree structure such as this, with each node having exactly two descendants, is called a binary tree. In general, trees are not restricted in the number of descendants each node may have.

Another useful recursive data structure is a linked list. It is similar to the tree structure, except that each node has only one access component. For example, the card catalog of a library system could be represented as a linked list. The first node of the list would represent the card associated with the book whose author's name is alphabetically first. The access part of this node contains a single pointer to the next card (next author's name in alphabetical order) in the catalog.

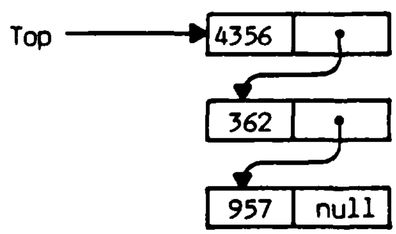
Pictorially the card catalog's representation would look like:



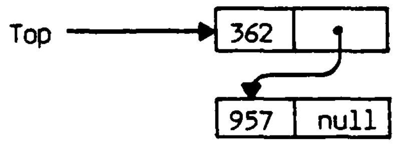
Popular uses of linked lists are to implement stacks and queues. A stack is a linked list in which items are added to (pushed on) or deleted from (popped off) the beginning of the list (top of the stack). This is the last-in, first-out method of item processing. Pushing the number 4356 onto the stack depicted as,



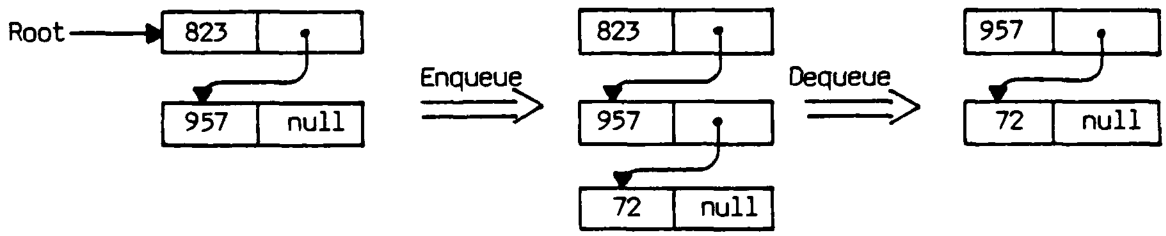
yields the stack depicted as follows:



Popping the stack yields the value 4356 and returns the stack to its earlier state:



Queues are first-in first-out stores. They may be represented as a linked list in which items are added to the end of the structure and deleted from the beginning of the queue. Pictorially it looks like:



There are other variations of the basic linked list, such as circular lists, doubly linked lists, and priority queues. The circular list is a linked list where the last node points to the beginning of the list. A doubly linked list is a linked list in which each node not only points to its successor, but to its predecessor as well. A priority queue as a linked list in which the nodes are processed in a first-in, highest-priority-out manner.

Having briefly described recursive types, let us now look at how to represent recursive types in Ada. As seen in Exercise 2.2, Ada has access types which act as pointers. These access types can be used (with some other Ada features) to represent recursive types.

To illustrate this, we will take the person example and show how it would be implemented in Ada. Ideally, we would like the structure for representing a person to look like this:

```
type Person is
    record
        Name       : String (1 .. 25);
        Birth_Date : Date;
        .
        .

        Mother     : Person;
        Father     : Person;
    end record;
```

This is not directly representable in Ada because the components Mother and Father cannot be of type Person. That is the type we are attempting to define! However, an access to this type could be used as the type of Mother and Father, which would then point to other objects of type Person. Still, there is a problem expressing this. When you write:

```
type New_Person is access Person;        -- **ILLEGAL
type Person is
    record
        .
        .
        Mother : New_Person;
        Father : New_Person;
    end record;
```

the declaration of New_Person is illegal because Person is not defined yet. Also, when you have:

```
type Person is
    record
        .
        .
        Mother : New_Person;            -- **ILLEGAL
        Father : New_Person;            -- **ILLEGAL
    end record;
type New_Person is access Person;
```

the declaration of Mother and Father are illegal because New_Person is not yet defined. Regardless of the order of these two type declarations, one of them will be used before it is declared.

What we need is some mechanism to say, "There is going to be a type called Person and the details of Person will come later. In the mean time we can define an access type pointing to a Person and it will be okay."

Ada has a feature called the incomplete type declaration which does exactly this. The incomplete type declaration must be followed eventually by an ordinary full type declaration for the same type. Before the full type declaration, the incomplete type can only be used as the designated type in an access type declaration. Only after the full type declaration (where the details of the type are specified) can the type be used as is any other type. (Thus, the only use for incomplete type declarations is in defining recursive types.)

Now a person can be represented in Ada as:

```
type Person;                          -- Incomplete type declaration
type New_Person is access Person;     -- used as the designated type
                                      -- of an access type declaration.
type Person is
    record
        .
        .
        Mother : New_Person;
        Father : New_Person;
    end record;
```

All recursive types can be implemented in this manner. Now that we can represent recursive structures in Ada, let us take a look at how we would use them.

Certain aspects of managing recursive data structures are common to all recursive types. The following examples will be illustrated in terms of a queue representation. The handling of other recursive types can be extrapolated from these examples.

```
type Queue_Node;
type Node_Pointer_Type is access Queue_Node;
type Queue_Node is
    record
        Data      : Natural;
        Next_Node : Node_Pointer_Type;
    end record;

Queue : Node_Pointer_Type := new Queue_Node;
```

The object Queue is an access to a record object with two components: Data (an integer value) and Next_Node (another access value). Note that at

this point, the value of Data is undefined and the value of Next_Node is null (i.e., it does not point to anything).

Recall that in order to assign values to the designated object we reference the object through Queue, such as:

```
Queue.Data := 6;
```

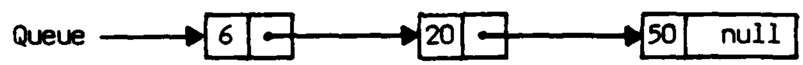
and

```
Queue.Next_Node := new Queue_Node;
```

Now to assign values to the new node, we would say:

```
Queue.Next_Node.Data      := 20;
Queue.Next_Node.Next_Node := new Queue_Node := (50, null);
```

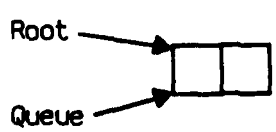
Pictorially Queue would look like:



Note that as we continue to add new nodes to the queue, the complexity of determining where to add the node increases. We need some mechanism for marking the end of the queue. In the following, we use two access values, Root (which "points to" the front of the queue) and Queue (which "points to" the end of the queue).

```
type Queue_Node;
type Node_Pointer_Type is access Queue_Node;
type Queue_Node is
    record
        Data : Natural;
        Next_Node : Node_Pointer_Type;
    end record;

Root  : Node_Pointer_Type := new Queue_Node;
Queue : Node_Pointer_Type := Root;
```

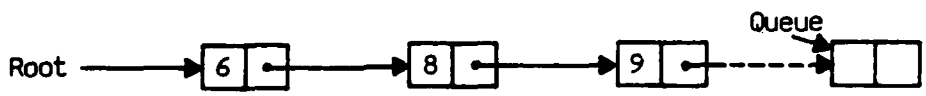
Pictorially, we start with two access values "pointing to" the same object.



Now when we add nodes, as in the following:

```
Queue.Data := X;
Queue.Next_Node := new Queue_Node;
Queue := Queue.Next_Node;
```

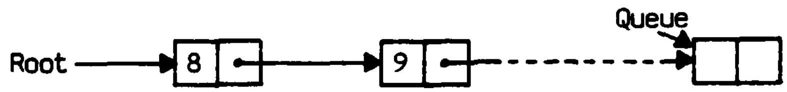
After several iterations we have, for example:



Note that the queue always has a "spare" or "dummy" element, which is the one pointed to by Queue. As we will see, this eliminates the need for treating the empty queue as a special case.

Now to delete the first element we simply change where Root points to.

```
Root := Root.Next_Node;
```



Note that the storage for the old first node can now be reclaimed. The deletion of some nodes, such as tree nodes, can be somewhat more complex because all nodes pointing to the node to be deleted have to be located before the deletion can be done. However, it is always possible to find the node by starting at the root and tracing forward to the node to delete. Doubly linked lists and trees in which each node points to its parent as well as to its descendants could be used for large structures subject to frequent deletions.

Other operations which may be performed on recursive data structures are sorting and searching. These operations will be addressed in Chapter 5 of this workbook.

Problem

A manufacturing company has a problem keeping track of when to reorder its supplies. Supply reorders arrive from different departments within the company and are processed when they arrive at the purchasing center. The supply orders are processed in order of priority. The purchasing department wants an automated system to keep track of outstanding reorder requests. Develop a system that keeps track of the reorder information:

Name of item
Manufacture of item
Cost
Reorder amount
Priority

and determines the supply order which is to be processed next.

Solution and Discussion

The first design problem to ask, when approaching this problem is, "How do we form an interface with the user of this system?" At first glance it looks as if the only users would be the people in purchasing who enter and process supply orders. However, in the future the whole re-ordering process will probably be automated. We therefore do not want to customize this system by connecting it to a user at a terminal. Instead, we want to make it as general and adaptable as possible. The obvious approach then is to make the system a package which can be used by other program units. (The other program unit would be the program which communicates with an actual person at a terminal).

Two subprograms are required: one which adds the supply order information, and another which returns the next order to be processed. The first subprogram requires two items: the supply order information and its priority. The second subprogram takes no parameters, and returns only the supply order information.

The data items needed are defined below:

```
subtype Name_Type is String (1 .. 15);     -- For name of item
                                           -- and manufacturer.
type Money_Type is range 0 .. 1E8;         -- Cost in cents.
type Priority_Type is range 1 .. 10;       -- Priority of order.

type Supply_Order is                       -- Type for a single order.
    record
        Item_Name    : Name_Type;
        Manufacturer : Name_Type;
        Cost         : Money_Type;
        Amount       : Positive;           -- Quantity of item.
    end record;
```

Using these types we can define the subprogram interfaces and combine it all into the package specification.

```
package Supply_Order_Manager is

    subtype Name_Type is String (1 .. 15);
    type Money_Type is range 0 .. 1E8;
    type Priority_Type is range 1 .. 10;
```

```
    type Supply_Order_Type is
        record
            Item_Name    : Name_Type;
            Manufacturer : Name_Type;
            Cost         : Money_Type;
            Amount       : Positive;
        end record;

    procedure Add_Supply_Order (Supply_Order : in Supply_Order_Type;
                                Priority     : in Priority_Type);

    function Order_To_Process return Supply_Order_Type;

end Supply_Order_Manager;
```

The next step is to decide how the supply order information is stored. As in Exercise 1.2 we need an external file for storing the orders. This aspect of the solution has been seen before so we will not spend time discussing it. A more important and pertinent question is, "How do we internally store the information?"

We could declare an array to represent a table as in Exercise 1.2, but note that in this problem additions and deletions are not just occasional, and that the quantity of information will decrease as well as increase. For the varying size we could allocate enough room for say 100, or 1000 orders, but as illustrated in Exercise 2.2, this is unnecessary. We can use access types.

But what do we access? We could access an array, as we did a table in Exercise 2.2. But we do not want a table structure because of the deletion of orders in the system (i.e., when they are processed). Orders are filled according to priority, rather than simply from the beginning or end. Therefore, we must "close the gap" left by a deletion, a fact which suggests a recursive structure such as the following:

```
type Queue_Node_Type;
type Node_Pointer_Type is access Queue_Node_Type;
type Queue_Node_Type is
    record
        Supply_Info : Supply_Order_Type;
        Priority    : Priority_Type;
        Next_Node   : Node_Pointer_Type;
    end record;
```

Having defined the elementary structure, we must now build it so that when the package is "with"ed, the order data is available to be processed, just as we did in Exercise 1.2. This is done within the package body:

```
with Sequential_IO;
package body Supply_Order_Manager is

    type Queue_Node_Type;
    type Node_Pointer_Type is access Queue_Node_Type;
    type Queue_Node_Type is
        record
            Supply_Info : Supply_Order_Type;
            Priority    : Priority_Type;
            Next_Node   : Node_Pointer_Type;
        end record;

    Root  : Node_Pointer_Type := new Queue_Node_Type; -- Queue and Root
    Queue : Node_Pointer_Type := Root;                -- point to a new
                                                      -- node.
    package Order_IO is new Sequential_IO (Queue_Node_Type);
    use Order_IO;

    Order_File : File_Type;
    File_Name  : constant String := "Supply_Orders.Dat";

    procedure Add_Supply_Order
            (Supply_Order : in Supply_Order_Type;
             Priority     : in Priority_Type) is separate;

    function Order_To_Process return Supply_Order_Type is separate;

begin  -- Supply_Order_Manager

    Open (Order_File, In_File, File_Name);

    while not End_of_File (Order_File)
    loop
        Read (Order_File, Queue.all);              -- New node filled.
        Queue.Next_Node := new Queue_Node_Type;    -- Allocate another.
        Queue := Queue.Next_Node;                  -- Queue pointed to
    end loop;                                      -- new dummy node.

    Close (Order_File);

end Supply_Order_Manager;
```

Note that the structure is built as an ordinary queue, with each new record added to the end of the structure. Initially, Root and Queue point to a newly allocated node of Queue_Node_Type. Then, in the loop, that node (Queue.all) is filled with data from the external file; a pointer to a new allocated node is put in its access part; and that pointer is assigned to Queue. When the package has finished executing, Queue points to the last allocated node (the "dummy" node). So whenever Supply_Order_Manager is imported, the queue configuration and pointers Root and Queue are such that new orders can easily be added to the queue, and order data can easily be processed.

Now we can take a look at the subprograms stubbed out in Supply_Order_Manager. The first subprogram, Add_Supply_Order, must fill the dummy node with the information associated with the new order, and append a new node to the end of the structure:

```
separate (Supply_Order_Manager)
procedure Add_Supply_Order (Supply_Order : in Supply_Order_Type;
                            Priority     : in Priority_Type) is

begin -- Add_Supply_Order

    Queue.all := (Supply_Order, Priority, new Queue_Node_Type);
    Queue := Queue.Next_Node;

end Add_Supply_Order;
```

The next subprogram, Order_To_Process, is more interesting. It must search through the queue to identify the supply order with the highest priority, delete that order from the queue, and return the supply information. To search the queue, we declare an access object of type Node_Pointer_Type to step through the queue. It starts at the Root, and is finished when it points to the same node as Queue. Queue points to the dummy node that is at the end of the queue.

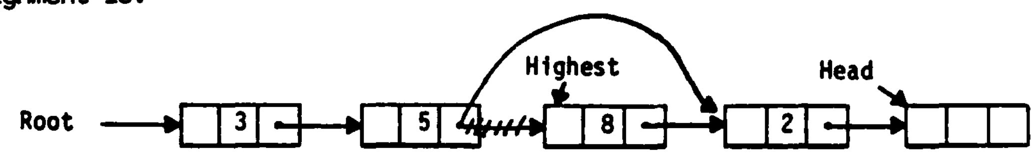
But before the queue can be searched, it must be ascertained that there is at least one order in it. To do this we can simply check if Root and Queue point to the same node, and if they do, raise an exception.

The following algorithm searches the queue, leaving an access object to identify the node with the highest priority:

```
Temp    : Node_Pointer_Type := Root.Next_Node;
Highest : Node_Pointer_Type := Root;

if Root = Queue then
    raise No_More_Orders;
else
    while Temp /= Queue
    loop
        if Highest.Priority < Temp.Priority then
            Highest := Temp;
        end if;
        Temp := Temp.Next_Node;
    end loop;
end if;
```

Notice that if several nodes in the queue have the highest priority the oldest order (first into the queue) will be chosen. Also, the exception No_More_Orders must be declared in the specification of Supply_Order_Manager.

When the loop has finished executing, the node with the highest priority (i.e. the one pointed to by Highest) must be removed from the queue. Essentially, we want the node before the highest priority node to point to the one after the highest priority node. Pictorially, the effect of this assignment is:



We first locate the previous node, and then change its Next_Node component to the value of the highest priority node's Next_Node component. The algorithm is essentially the following:

```
Temp := Root;

-- Search until Temp points to the node preceeding the
-- node with the highest priority.

while Temp.Next_Node /= Highest
loop
    Temp := Temp.Next_Node;
end loop;

-- Make this node's access component point to the node
-- following the node with the highest priority.

Temp.Next_Node := Highest.Next_Node;
```

We have assumed that the highest priority node is not the first node in the queue. In that case we must set Root to the next node in the queue. The complete algorithm follows:

```
-- when highest is the first node in the queue

if Highest = Root then

    Root := Highest.Next_Node;

else

    Temp := Root;

    -- Search until Temp points to the node preceeding the
    -- node with the highest priority.

    while Temp.Next_Node /= Highest
    loop
        Temp := Temp.Next_Node;
    end loop;

    -- Make this node's access component point to the node
    -- following the node with the highest priority.

    Temp.Next_Node := Highest.Next_Node;

end if;
```

The rest of the function is straightforward. It appears at the end of this exercise with the complete solution.

Note that this approach assumes that at any given time the number of orders waiting to be processed will be rather small. Should this not be the case in the future, this structure could be modified to a doubly linked list. (Again, we are assuming that this is unnecessary, but be aware that the type and the algorithm that builds the structure would have to change in order to implement a doubly linked list.)

We must now ask ourselves, "Is the system complete?" Although we have completed the actual requirements, the answer is no because we have not built in any facility for storing the information permanently. If the current solution were executed, all changes made to the queue during that execution

would be lost when it finished. Recall that our external file, which originally loaded the structure, has not been modified. And so the next time the package is loaded, the same structure will be built.

How do we solve this? We could update the external file with each change, but this is expensive in terms of the I/O required. Also, as in Exercise 1.2, in order to add to the end of the file we would have to switch to Direct_IO.

But if we are providing a new procedure to signal end of processing, it might as well be one which stores the queue in its final form, obviating the expensive intermediate updates. Such a procedure would declare a temporary access object, just as in the procedure that searched for the order with the highest priority. The temporary object would move through the queue, from the Root to Queue, writing each successive node to the external file. The code follows:

```
separate (Supply_Order_Manager)
procedure Store_Supply_Orders is

    Order : Node_Pointer_Type := Root;

begin -- Store_Supply_Orders

    Open (Order_File, Out_File, File_Name);

    while Order /= Queue
    loop
        Write (Order_File, Order.all);  -- Write node to external file.
        Order := Order.Next_Node;       -- Move to next file.
    end loop;

    Close (Order_File);

end Store_Supply_Orders;
```

By making this procedure available to the users so that they can signal when they are done, we have also given them the ability to "backup" their work at any time. Although the intent of the procedure was to signal that they are done, calling that procedure has no effect on program execution.

Now our system is complete. The code for the entire system follows:

```
package Supply_Order_Manager is

    subtype Name_Type is String (1 .. 15);
    type Money_Type is range 0 .. 1E8;          -- In cents.
    type Priority_Type is range 1 .. 10;

    type Supply_Order_Type is
        record
            Item_Name    : Name_Type;
            Manufacturer : Name_Type;
            Cost         : Money_Type;
            Amount       : Positive;
        end record;

    No_More_Orders : exception;

    procedure Add_Supply_Order (Supply_Order : in Supply_Order_Type;
                                Priority     : in Priority_Type);

    function Order_To_Process return Supply_Order_Type;

    procedure Store_Supply_Orders;

end Supply_Order_Manager;


with Sequential_IO;
package body Supply_Order_Manager is

    type Queue_Node_Type;
    type Node_Pointer_Type is access Queue_Node_Type;
    type Queue_Node_Type is
        record
            Supply_Info : Supply_Order_Type;
            Priority    : Priority_Type;
            Next_Node   : Node_Pointer_Type;
        end record;

    Root  : Node_Pointer_Type := new Queue_Node_Type; -- Queue and Root
    Queue : Node_Pointer_Type := Root;                -- point to a new
                                                      -- node.
    package Order_IO is new Sequential_IO (Queue_Node_Type);
    use Order_IO;

    Order_File : File_Type;
    File_Name  : constant String := "Supply_Orders.Dat";
```

```
    procedure Add_Supply_Order
            (Supply_Order : in Supply_Order_Type;
             Priority     : in Priority_Type) is separate;

    function Order_To_Process return Supply_Order_Type is separate;

    procedure Store_Supply_Orders is separate;

begin  -- Supply_Order_Manager

    Open (Order_File, In_File, File_Name);

    while not End_of_File (Order_File)
    loop
        Read (Order_File, Queue.all);               -- Fill node.
        Queue.Next_Node := new Queue_Node_Type;     -- Allocate another.
        Queue := Queue.Next_Node;                   -- Queue pointed to
    end loop;                                       -- new node.

    Close (Order_File);

end Supply_Order_Manager;

separate (Supply_Order_Manager)
procedure Add_Supply_Order (Supply_Order : in Supply_Order_Type;
                            Priority     : in Priority_Type) is

begin -- Add_Supply_Order

    Queue.all := (Supply_Order, Priority, new Queue_Node_Type);
    Queue := Queue.Next_Node;

end Add_Supply_Order;

separate (Supply_Order_Manager)
function Order_To_Process return Supply_Order_Type is

    Temp    : Node_Pointer_Type := Root.Next_Node;
    Highest : Node_Pointer_Type := Root;

begin    -- Order_To_Process

    if Root = Queue then
        raise No_More_Orders;
    else
        while Temp /= Queue
        loop
            if Highest.Priority < Temp.Priority then
                Highest := Temp;
            end if;
            Temp := Temp.Next_Node;
        end loop;
```

```
            --  when highest is the first node in the queue

            if Highest = Root then

                Root := Highest.Next_Node;

            else

                Temp := Root;

                --  Search until Temp points to the node preceding the
                --  node with the highest priority

                while Temp.Next_Node /= Highest
                loop
                    Temp := Temp.Next_Node;
                end loop;

                --  Make this node's access component point to the node
                --  following the node with the highest priority

                Temp.Next_Node := Highest.Next_Node;

            end if;

        end if;

    end Order_To_Process;

    separate (Supply_Order_Manager)
    procedure Store_Supply_Orders is

        Order : Node_Pointer_Type := Root;

    begin -- Store_Supply_Orders

        Open (Order_File, Out_File, File_Name);

        while Order /= Queue
        loop
            Write (Order_File, Order.all);
            Order := Order.Next_Node;
        end loop;

        Close (Order_File);

    end Store_Supply_Orders;
```

CHAPTER 3

DATA ABSTRACTION

EXERCISE 3.1

DISCRIMINANTS: STRUCTURE, USE AND DEFAULT VALUES

Objective

The Tutorial in Exercise 1.1 alluded to discriminants when explaining that a record with a discriminant is the only composite type that may have components of an unconstrained type. This chapter will focus on the meaning and use of discriminants in more depth.

Tutorial

Discriminants are record components with special properties. Loosely speaking, they act as parameters to record type declarations. They may be used to express the dependence of one or more record components on some other component. The Ada language rules for discriminants affect both record types and objects of those types; this Tutorial will first discuss the use of discriminants in type declarations and then their impact on objects.

Record Types With Discriminants

Consider the following example. A message consists of several parts, as outlined below:

- a routing indicator
- a security classification
- a transmission mode (synchronous or asynchronous)
- for synchronous transmissions, whether even or odd parity is to be computed
- text

At first glance, it seems straightforward to encapsulate message components in a single data structure:

```
subtype Routing_Indicator_Type is String (1 .. 3);

type Classification_Type is (Unclassified, Confidential,
                             Secret, Top_Secret);
```

```
type Transmission_Mode_Type is (Asynchronous, Synchronous);

type Parity_Type is (Even, Odd);

subtype Character_Count_Type is Natural range 0 .. 127;

subtype Text_String is String (1 .. Character_Count_Type'Last);

type Text_Type;

type Text_Pointer_Type is access Text_Type;

type Text_Type is
    record
        Text      : Text_String;
        Next_Text : Text_Pointer_Type;
    end record;

type Message_Type_1 is
    record
        Routing_Indicator       : Routing_Indicator_Type;
        Security_Classification : Classification_Type;
        Transmission_Mode       : Transmission_Mode_Type;
        Parity                  : Parity_Type;
        Text                    : Text_Pointer_Type;
    end record;
```

The above data structure fails to reflect the fact that parity only applies to synchronous transmissions. Whatever value is stored in the parity component is meaningless when the mode is asynchronous. In Ada, the way to express this dependence explicitly is to make the Transmission_Mode component a discriminant component. The presence of a Parity component is then tied to the value of the Transmission_Mode component, as shown in the record type declaration below:

```
type Message_Type_2 (Transmission_Mode : Transmission_Mode_Type) is
    record
        Routing_Indicator       : Routing_Indicator_Type;
        Security_Classification : Classification_Type;
        Text                    : Text_Type;
        case Transmission_Mode is
            when Synchronous =>
                Parity : Parity_Type;
            when Asynchronous =>
                null;
        end case;
    end record;
```

This record type declaration sets up a record with either four or five components, depending on the value of the first component, the discriminant Transmission_Mode. When this component has the value Synchronous, objects of type Message_Type_2 will have a total of five components: Transmission_Mode, Routing_Indicator, Security_Classification, Text, and Parity. When the component Transmission_Mode has the value Asynchronous, then objects of type Message_Type_2 only have four components: Transmission_Mode, Routing_Indicator, Security_Classification, and Text. The component Transmission_Mode is called the discriminant of the record type Message_Type_2. The portion of the record type declaration beginning with case and ending in end case is called the variant part of the record. Notice also the use of the reserved word null in the variant part. Just as in case statements there must be one alternative for each possible value of the case expression, so in the variant part there must be one set of components for each possible value of the discriminant. In order to indicate that the set of components is empty, it is necessary to specify a null component list as shown above.

For another example, let us impose an additional requirement on the above message data structure: a message can have multiple routing indicators, but the number is not known a priori. The logical modification to the data structure is to define an array type to hold the routing indicators:

```
type Routing_Indicator_List_Type is
      array (Positive range<> ) of Routing_Indicator_Type;
```

Next, we would want to modify the type of the Routing_Indicator component of the Message_Type_2 record to be the newly defined Routing_Indicator_List_Type. A record component of an array type, however, must be of a constrained array subtype, and Routing_Indicator_List_Type is unconstrained. We could choose an arbitrary upper bound to accommodate the largest expected number of routing indicators, but this solution fails to reflect that different messages may have different numbers of routing indicators. Alternatively, we can supply this information through a discriminant, as illustrated below:

```
type Message_Type_3
        (Routing_Indicator_Count : Positive;
         Transmission_Mode       : Transmission_Mode_Type) is
    record
        Routing_Indicator :
            Routing_Indicator_List_Type (1 .. Routing_Indicator_Count);
        Security_Classification : Classification_Type;
        Text                    : Text_Type;
        case Transmission_Mode is
            when Synchronous =>
                Parity : Parity_Type;
            when Asynchronous =>
                null;
        end case;
    end record;
```

In the above record type declaration, we have introduced a second discriminant, Routing_Indicator_Count, of type Positive, with which we then constrain the Routing_Indicator component's unconstrained array type, Routing_Indicator_List_Type.

Thus we see a second use for discriminants, namely to provide a constraint for an array type component of a record type. In addition to constraining array type components of record types, a discriminant can be used to constrain a component which itself is a record type with discriminants. An example of this second situation is given later in the tutorial, during the discussion of record objects. In fact, a useful paradigm to remember is that one way to create a variable length array is through a record type with at least two components: the first component is the discriminant whose value will be the length of the array; the second component is the unconstrained array type whose constraint is provided by the discriminant. Recall that this approach is not the only one available; another implementation of variable length arrays uses access types.

Like case expressions and array indices, discriminants must belong to a discrete type, (that is any enumeration or integer type). This rule is consistent with the rules for the type of array indices and for the type of a case expression.

Like other record components, discriminants may be given default values. A default value, however, may not be given for one discriminant of a record type unless it is given for all of them This rule differs from the rule for assigning default values to simple record components (i.e. components which are not discriminants), where the user may provide default values for an arbitrary subset of the record components. For example, the Routing_Indicator_Count discriminant of Message_Type_3 may be assigned a default value of 5:

```
type Message_Type
        (Routing_Indicator_Count : Positive := 5;
         Transmission_Mode       : Transmission_Mode_Type :=
                                          Synchronous) is
    record
        Routing_Indicators :
            Routing_Indicator_List_Type (1 .. Routing_Indicator_Count);
        Security_Classification : Classification_Type;
        Text                    : Text_Type;
        case Transmission_Mode is
            when Synchronous =>
                Parity : Parity_Type;
            when Asynchronous =>
                null;
        end case;
    end record;
```

Notice that both discriminants were assigned default values, even though the original intention was just to provide a default value for the Routing_Indicator_Count component.

At this point, a summary of the syntax for record type declarations with discriminants is in order. The general pattern is shown rather than a list of all the variations possible.

```
type Identifier ( discriminant_part ) is
    record

        component_list;

        variant_part;     -- if any

    end record;
```

where the Variant_Part is structurally similar to the case statement:

```
case Discriminant_Value is
    when Value_1 =>
        component_list;
        nested_variant_part;  -- if any
    when Value_2 =>
        component_list;
        nested_variant_part;  -- if any
    -- etc. such that ALL possible values of the
    --  discriminant are accounted for
end case;
```

The discriminant part is the declaration of the names of the discriminants, their types, which must be discrete types, and their default values, if any. Either all or none of the discriminants must have default values. The Component_List is a list of component names and their types, with optional default values. The component type may use a discriminant to provide a required constraint. The variant part, if it exists, is a single case structure. Within the variant part, nested variants are allowed. A variant is the set of components which belong to a record type for a value or set of values of a discriminant. If there are no components of the record for a particular discriminant value(s), then the null component must be supplied, as in the example below:

```
case Discriminant is
    when Value =>
        Component : Component_Type;
    when others =>
        null;
end case;
```

Objects of Types with Discriminants

Given a record type with discriminants, special considerations apply to the declaration of objects of the type. If default values have not been

specified for the discriminants, then each object declaration must specify a discriminant constraint. Specifying the discriminant constraint in the object declaration is analogous to providing the bounds of an unconstrained array when the array object is declared. Record objects are declared as:

```
Message_1 : Message_Type_3 (1, Asynchronous);
Message_2 : Message_Type (1, Asynchronous);
```

or, equivalently, using named notation:

```
Message_2 : Message_Type (Routing_Indicator_Count => 1,
                          Transmission_Mode => Asynchronous);
```

When discriminant constraints are supplied in the object declaration then the object is said to be a constrained object; the values of the discriminants for that particular object are fixed. If, on the other hand, the type declaration provides default values for the discriminant, then either constrained or unconstrained objects may be declared. For a constrained object, discriminant constraints are provided when the object is declared, overriding the default values and permanently fixing the values of the discriminants. For an unconstrained object, no discriminant constraints are specified in the object declaration, and the values of the discriminants are initially the default values, but they may be changed at any time. Thus, for the declaration

```
Message_3 : Message_Type;
```

the following would be initially true:

```
Message_3.Routing_Indicator_Count = 5    -- True
Message_3.Transmission_Mode = Synchronous  -- True
```

Because no constraints are specified when the object is declared, this object is said to be unconstrained. If the discriminants are given default values but the programmer only wishes to change the value of one of them, then he must nevertheless restate the values for all the discriminants, as in

```
Message_4 : Message_Type (Routing_Indicator_Count => 5,
                          Transmission_Mode => Asynchronous);
```

A constrained object is constrained for life. When an object is unconstrained, then the value of its discriminant may be changed subject to the following restrictions. First, this object must itself be a variable. Secondly, the only way to change the value of the discriminant is by assigning a new value to the entire record. Direct assignment as well as use of a discriminant as a parameter of mode out or in out is forbidden. For example, consider the three objects declared above, Message_2, Message_3, and Message_4. Only the object Message_3 may be assigned an aggregate whose Routing_Indicator_Count and Transmission_Mode components may override the default values provided for these discriminants. In order to change the value of the discriminants, whole record assignment must be used:

```
Text_Pointer : Text_Pointer_Type := new Text_Type'
              ("Test sequence loaded" &
                  (21 .. Text_String'Last => ' '),
               null);
Message_3 := (Routing_Indicator_Count => 2,
              Transmission_Mode       => Synchronous,
              Routing_Indicators      => ("X2T","88N"),
              Security_Classification => Secret,
              Text                    => Text_Pointer,
              Parity                  => Even);
```

Furthermore, if changing the value of a record component which depends on a discriminant also requires changing the value of the discriminant, as in assigning a third routing indicator to the Routing_Indicators component of Message_3, then whole record assignment must be used (provided of course that the discriminant has a default value and the object in question is unconstrained.) In summary, if the programmer wants the capability to change the value of the discriminant of a record object, then

- the corresponding type declaration must specify default initial values for each discriminant, and
- the object itself must be unconstrained, i.e. no discriminant constraints are imposed when this object is declared.

There are storage considerations involved in declaring default values for the discriminants of a record type. For a given object, some implementations will allocate enough storage to accommodate a value corresponding to the largest possible values for its discriminants. Thus if the largest positive number on the machine is 32,767 and an unconstrained Message_Type object is declared, then in creating this object, the system might attempt to allocate space for 32,767 three-character strings for the Routing_Indicators component. This attempt will likely raise the predefined exception Storage_Error. Care should be taken, therefore, in the use of discriminants with default values. (There is a simple solution to the above problem. A subtype with a smaller range, say 127, should be declared and used instead of the subtype Positive.)

It was mentioned earlier that a component of a record can be of another record type with discriminants. This component is treated analogously to objects of record types with discriminants. Recall that for objects, if no default values are provided for the discriminants in the type declaration, then the discriminant constraints must be provided in the object declaration. The same holds true for components whose type is a record with discriminants. If the discriminants are declared with default values, then no further constraints need be specified when this component is declared. If, however, no default values are specified, then the component must be constrained. An example follows:

```
type R1 (D1 : Positive := 1) is
    record
        R1_C1 : String (1 .. D1);
    end record;

type R2 (D2 : Positive) is
    record
        R2_C1 : String (1 .. D2);
    end record;

type R3 is
    record
        R3_C1 : R1;
        R3_C2 : R2 (D2 => 10);
    end record;
```

Another application of record types whose discriminants have default values is in the component type of an array type. For example,

```
type A1 is array (1 .. 4) of R1;
```

declares an array type whose components are records of varying length strings. It would not be feasible to declare an array whose component type was R2 without specifying a discriminant constraint for D2:

```
type A2 is array (1 .. 10) of R2 (D2 => 4);
```

Thus, when the discriminants are given default values, the programmer can essentially create an array of dissimilar objects, as shown below:

```
O1_A1 : A1 := ((8, "Murphy's"), (4, "laws"),
               (2, "on"), (10, "technology"));
```

It is also possible to use the discriminant of a record type in the discriminant constraint of a component record type. Suppose that the requirements of the message type are further modified to state that not only are there multiple routing indicators per message, but also that each set of routing indicators must be tagged with the unique identification number of the message. Thus, the routing indicators component is now a record type:

```
type Message_ID_Type is range 1 .. 1E10;

type Routing_Indicators_Set_Type
      (Routing_Indicator_Count : Positive) is
   record
      Message_ID          : Message_ID_Type;
      Routing_Indicators :
         Routing_Indicator_List_Type.(1 .. Routing_Indicator_Count);
   end record;
```

The record defining a message type must also be modified:

```
        type Message_Type_4
                 (Routing_Indicator_Count : Positive := 5;
                  Transmission_Mode : Transmission_Mode_Type := Synchronous) is
            record
                Routing_Indicators        :
                    Routing_Indicators_Set_Type (Routing_Indicator_Count);
                Security_Classification : Classification_Type;
                Text                      : Text_Type;
                case Transmission_Mode is
                    when Synchronous =>
                        Parity : Parity_Type;
                    when Asynchronous =>
                        null;
                end case;
            end record;
```

The discriminant Routing_Indicator_Count is declared in the record type declaration of Message_Type_4 and is then used to constrain the type of the component Routing_Indicators, Routing_Indicators_Set_Type, itself a record type with a discriminant.

Exercises 2.2 and 2.3 discussed access types. One of the important points to remember from those Tutorials is that there are no restrictions on what type an access type can point to. Therefore, it is perfectly natural to create an access type that points to a record type with discriminants. Instead of manipulating message objects, we can create an access type to point to a Message_Type record:

```
type Message_Pointer is access Message_Type;
Message_5 : Message_Pointer;
```

The pointer Message_5 can point to any message in the system. However, once a variable is allocated, then that variable is always constrained, regardless of whether default values were supplied for the discriminants or not. In other words, when a message is first allocated, this message will have five routing indicators and a synchronous transmission mode (assuming the default values are not overridden). These discriminant constraints will hold for the lifetime of this accessed object. To later change what Message_5 points to, so that it points to a message with ten routing indicators, synchronous

transmission mode and even parity, one cannot effect this change through a ".all" assignment. The solution is to change the space to which the access object points by allocating a new variable with the new constraints:

```
Message_5 := new Message_Type'
                (Routing_Indicator_Count => 10,
                 Transmission_Mode       => Synchronous);
```

If, however, we want to restrict the kind of messages to which a Message_Pointer object can point, then we can specify this restriction by constraining the access object (as opposed to the accessed object) when it is declared:

```
Message_6 : Message_Pointer
                (Routing_Indicator_Count => 3,
                 Transmission_Mode => Asynchronous);
```

Now, Message_6 can only point to messages with 3 routing indicators and an asynchronous transmission mode, and it cannot be allocated a message with a different set of discriminant constraints. The declaration of Message_6 implies that it belongs to a subtype of the access type Message_Pointer, such that this subtype defines a subset of the pointer values, consisting of pointers to records with a given discriminant constraint.

Problem

A central control panel is used to monitor different kinds of sensors throughout a one story building. Each sensor has a unique ten-character identification code. Within the building, different kinds of sensors can be found at a given location. The different kinds of sensors detect temperature above 75 degrees Fahrenheit, below 50 degrees Fahrenheit, humidity greater than 60%, smoke, and dust levels in excess of one part per 100,000. The smoke sensors are also heat sensitive. Sensors are located in the computer room, either on the east or west wall, and also in the one hallway, either on the east end, the west end, or the center. Should a sensor detect abnormal conditions, an alarm is set, registering the location on the central control panel.

Write a package containing the data structures necessary to describe this central control panel.

Solution and Discussion

In approaching this problem let us develop skeleton data types corresponding to the information in the problem statement.

Our goal is:

```
type Control_Panel_Type is ... ;
```

This structure will certainly be a composite type, either a record or an array. The record structure might seem like a logical approach to follow because the information on sensors is disparate: heat sensors and smoke sensors record different data; and different locations have different numbers and kinds of sensors. Because these numbers are not known a priori, we cannot specify one record component per sensor.

A more reasonable approach would be to have one component per location, or one per kind of sensor. When there is one component per location, then all the components of the record are of the same type because at all locations there can be any number of different kinds of sensors. When all the components of a record are of the same type, then a record structure is inappropriate, and an array structure should be used. This option is discussed further in the succeeding paragraphs.

The other method is to have one component per kind of sensor. This option is inelegant and does not reflect the requirements for the following reasons. Each sensor component would itself be a record type which would have to contain the information about the number of that kind of sensor at each possible location in the building. Most likely, this information would have to be represented using a record structure with a discriminant for the actual number of sensors at each location:

```
type Heat_Sensors_Array is (Positive range <>) of Heat_Sensor_Data;

type Heat_Sensor_Component
        (Heat_Sensor_Count_Computer_Room_East,
         Heat_Sensor_Count_Computer_Room_West,
         Heat_Sensor_Count_Hallway_East,
         Heat_Sensor_Count_Hallway_Center,
         Heat_Sensor_Count_Hallway_West : Natural := 0) is
    record
        Heat_Sensors_Computer_Room_East :
            Heat_Sensors_Array
                (1 .. Heat_Sensor_Count_Computer_Room_East);
        Heat_Sensors_Computer_Room_West :
            Heat_Sensors_Array
                (1 .. Heat_Sensor_Count_Computer_Room_West);
        Heat_Sensors_Hallway_East :
            Heat_Sensors_Array
                (1 .. Heat_Sensor_Count_Hallway_East);
        Heat_Sensors_Hallway_Center :
            Heat_Sensors_Array
                (1 .. Heat_Sensor_Count_Hallway_Center);
        Heat_Sensors_Hallway_West :
            Heat_Sensors_Array
                (1 .. Heat_Sensor_Count_Hallway_West);
    end record;
```

This solution is unnecessarily cumbersome and will not be pursued further.

An array structure would imply that the control panel is a collection of information on different sensors, such that the information on any particular sensor fits into a single data structure model. This method is possible because of the rules concerning discriminants with default values. To reiterate the relevant rule, if a record discriminant has a default value, then a discriminant constraint does not need to be provided when this record type is used as the component type of an array. In practice, if we encapsulate the data for any given sensor in a record type with discriminants with default values, we can in turn use this record type as the component type of the array type Control_Panel_Type:

```
type Location_Type is
        (Computer_Room_East, Computer_Room_West,
         Hallway_East, Hallway_Center, Hallway_West);

type Control_Panel_Type is array (Location_Type)
        of Control_Panel_Entry_Type;
```

Control_Panel_Entry_Type must be a record type indicating which sensors are at a location. Additionally, it must indicate whether one of the sensors set off an alarm. Because an alarm is either ringing or off, the appropriate representation is a Boolean type. The record type definition for Control_Panel_Entry_Type follows:

```
type Control_Panel_Entry_Type
      (Number_Of_Sensors : Natural := 0) is
   record
      Alarm_Set   : Boolean;
      Sensor_List : Sensor_List_Type (1 .. Number_Of_Sensors);
   end record;
```

The discriminant Number_Of_Sensors is given a default value of zero so that a constraint does not need to be specified in the array type declaration Control_Panel_Type. The declaration of Control_Panel_Type is appropriate in the context of the problem statement; it is concise and uncluttered. For each location in the building there is a Control_Panel_Entry_Type component which indicates whether any sensors are present at that location, whether the alarm at that location is ringing, and if there are sensors, a list of the data for those sensors.

The Sensor_List component is an array of sensor data. While some of the data for sensors differs, all have one component in common, namely their unique identification code, implemented as a string. The rest of the data is captured through the variant part, where for each kind of sensor, there is a variant. The discriminant in this case is the name of the sensor, represented through an enumeration type.

```
type Sensors_Type is
        (Heat_Sensor, Cold_Sensor, Humidity_Sensor,
         Smoke_Sensor, Dust_Sensor);

subtype Serial_Number_Type is String (1 .. 10);

type Sensor_Data_Type
        (Sensor : Sensors_Type := Smoke_Sensor) is
    record
        Sensor_ID : Serial_Number_Type;
        case Sensor is
            when Heat_Sensor | Cold_Sensor =>
                Temperature : Temperature_Type;
            when Humidity_Sensor =>
                Humidity : Humidity_Type;
            when Smoke_Sensor =>
                Smoke_Present : Boolean;
                Temperature   : Temperature_Type;
            when Dust_Sensor =>
                Dust_Level : Dust_Level_Type;
        end case;
    end record;
```

The complete solution code follows.

```
package Control_Panel is

    type Sensors_Type is
            (Heat_Sensor, Cold_Sensor, Humidity_Sensor,
             Smoke_Sensor, Dust_Sensor);

    type Location_Type is
            (Computer_Room_East, Computer_Room_West,
             Hallway_East, Hallway_Center, Hallway_West);

    type Temperature_Type is digits 4 range -883.4 .. 10_000.0;
        -- degrees Fahrenheit
        -- -883.4 corresponds roughly to 0 degrees Kelvin

    type Humidity_Type is digits 4 range 0.0 .. 100.0;

    type Dust_Level_Type is range 0 .. 100_000; -- in parts per 100_000

    subtype Serial_Number_Type is String (1 .. 10);

    type Sensor_Data_Type (Sensor : Sensors_Type := Smoke_Sensor) is
        record
            Sensor_ID : Serial_Number_Type;
            case Sensor is
                when Heat_Sensor | Cold_Sensor =>
                    Temperature : Temperature_Type;
                when Humidity_Sensor =>
                    Humidity : Humidity_Type;
                when Smoke_Sensor =>
                    Smoke_Present : Boolean;
                    Temperature : Temperature_Type;
                when Dust_Sensor =>
                    Dust_Level : Dust_Level_Type;
             end case;
        end record;

    type Sensor_List_Type is array (Positive range<>)
            of Sensor_Data_Type;

    type Control_Panel_Entries
             (Number_Of_Sensors : Positive := 1) is
        record
            Alarm_Set   : Boolean;
            Sensor_List : Sensor_List_Type (1 .. Number_Of_Sensors);
        end record;

    type Control_Panel_Type is array (Location_Type)
            of Control_Panel_Entries;

end Control_Panel;
```

EXERCISE 3.2

PRIVATE AND LIMITED PRIVATE TYPES

Objective

This exercise discusses data abstraction in Ada and how private and limited private types are used to implement this programming philosophy.

Tutorial

A discussion of private and limited private types involves more than just a presentation of the syntactic and semantic rules. It also involves the concept of data abstraction. This tutorial will begin by discussing the ideas behind abstract data types in order to motivate the discussion of private types.

What is an abstract data type? The definition draws heavily on the basic definition of a data type. A type is defined to be a set of values, a set of operations on those values, and a set of relationships between those operations. For example, the predefined type Integer on a particular machine is defined to be the set of integer values, say -32,768 to 32,767. The set of operations for objects of type Integer would be the operations which the Ada language defines for integer types (addition, multiplication, subtraction, division, exponentiation, absolute value, remainder, modulus, and the relational operations). The set of relationships between the operations for Integer are those of precedence, also defined by the language, and the arithmetic axioms used in everyday arithmetic, such as:

```
I + 0 = I              -- additive identity
I + J = J + I          -- commutative
I * 1 = I              -- multiplicative identity
```

A single type declaration by itself does not necessarily constitute an abstraction. An abstraction is concerned not with the representation of data, but with its behavior. It may consist of many type declarations and subprogram declarations operating on these types. These declarations taken together constitute a primitive for the user. Consider for example the set

type discussed in Exercise 2.1. In manipulating a set the user wants to talk about the set, not an array of Booleans or a linked list. The actual representation of the set is irrelevant to the user, as long as the operations on sets behave as we expect them to. Neither the declaration of the enumeration type representing the elements of the universe set nor the declaration of a set type as an array of Booleans alone is sufficient to define the abstract idea of a set. It is the combination of these two types, the definition of the empty and universe sets, as well as the declaration of the basic operations of intersection, union, subsetting, difference, insertion and complement which constitute the abstraction because all of these are essential to describing the properties of sets.

In a good abstraction, the qualities or general characteristics of the object are conveyed, independent of the actual physical realization of this object. The designer builds a simple, clean interface which supports the expected behavior of the abstraction from the user's point of view. This approach is referred to as information hiding, data encapsulation and modularity. Information hiding refers specifically to the separation of the implementation from the abstraction. The idea here is that an abstraction could be implemented in more than one way, and that users of the abstraction should not rely on a particular implementation. When they do not rely on the implementation, then they are not affected by any modifications to it. Thus information hiding increases the maintainability of a program and the flexibility of the design. The potential problem witn the aforementioned set type is that if a user ignores the primitive set operations provided (such as union and intersection) and performs instead logical operations (i.e. **and** and **or**) on his sets, then his programs will be invalidated if the set package maintainer modifies the implementation of sets from an array of Booleans to a linked list. To prevent this problem, the package designer can avail himself of another Ada feature, namely private types, and he can thereby make the implementation of an abstraction inaccessible to the user of this abstraction. Through private types, Ada provides the designer and the user with a mechanism to enforce an abstraction and to distinguish between different levels of abstraction.

The other qualities of a good abstraction, modularity and data encapsulation are best described in terms of packaging. Encapsulating an abstraction means packaging its constituent types and operations into a single unit, which itself becomes the abstraction from the user's point of view. Ada packages correspond directly to this principle. Furthermore, because private types may only be declared inside a package, the related concepts of data abstraction, information hiding, and encapsulation are unified into a powerful programming technique.

Consider an implementation of text composed of an unknown number of total characters, to a maximum of 127 characters. Because of the variable length, the text is implemented through a record structure. Any individual block holds both text and the associated actual character count (to a maximum of 127). Operations on text include:

- create text from a string
- delete text
- append text to existing text
- compare two Text_Type objects for lexicographical ordering or equality
- insert text after a point in the existing text
- search for the presence of a string in the text

(Ignore exception conditions for now; this example will be expanded shortly to linked blocks of text to accommodate text longer than 127 characters. Assume also that all blocks are blank filled to ensure that equality behaves correctly.) Procedures and functions performing these operations must be defined on Text_Type and incorporated into the package specification:

```
package Text_Package is

    subtype Character_Count_Type is Integer range 0 .. 127;

    subtype Text_String is String (1 .. Character_Count_Type'Last);

    type Text_Type is
        record
            Character_Count : Character_Count_Type;
            Text            : Text_String;
        end record;

    procedure Append (To_Text   : in Text_Type;
                      More_Text : in out Text_Type);

    procedure Delete (Phrase : in String;
                      Text   : in out Text_Type);

    function "<" (Text_1, Text_2 : Text_Type) return Boolean;

    function ">" (Text_1, Text_2 : Text_Type) return Boolean;

    procedure Insert (More_Text    : in String;
                      After_String : in String;
                      To_Text      : in out Text_Type);

    function Is_in_String (Phrase : String; Text : Text_Type)
                          return Boolean;

end Text_Package;
```

Notice in the above package how the operators ">" and "<" are overloaded. The procedure Append adds text after the last character of the existing text, provided room is available. If the existing text is empty, then the text is added to this null block. The procedure Delete deletes a phrase from a block of text, if the phrase is found in this block of text. The procedure Insert inserts a phrase after the last character of the indicated text (the parameter After_String). The function Is_in_String returns a Boolean value depending on whether a phrase could be found in an existing piece of text.

The integrity of the text contained in a particular message can easily be compromised. For instance, the text of a particular block could be changed by a user who imports this package, while the character count is not updated to reflect the number of characters in the new version. Comparing two blocks for equality will no longer produce the expected results because of the

internal inconsistency of the record. In order to prevent such inconsistencies from occurring, we need to declare Text_Type in such a way that the user cannot make arbitrary changes. Because Text_Type should be visible to the user, it cannot be declared within a package body. Declarations within a package body are only visible inside the package body; they are not available to any program units which import this particular package. On the other hand, declaring Text_Type within a package specification exposes the implementation of the type to all potential users and abusers.

The solution is to use Ada's private type mechanism. The essence of the private type is to make the name of the type and some operations on it available to a user, while at the same time hiding the implementation details. Private types can be used only in restricted ways. These restrictions force the programmer to write implementation-independent, and therefore more maintainable code. Maintainability is further enhanced because there is now a simpler, clearly-defined interface. The package specification serves to enumerate the list of ways in which the package can interact with the outside world, simplifying the maintainer's job because he can clearly see the aspects of the abstraction that are relevant, and conversely that can be ignored by, the package users.

A private type is declared in the package specification:

```
type Text_Type is private;
```

All that is known at this point is that there is a type called Text_Type whose implementation is only known inside the package. The user may declare objects of type Text_Type and may assign objects to one another. Given the objects:

```
Text, Sentence : Text_Type;
```

the following is allowed:

```
Text := Sentence;
```

Other predefined operations allowed on Text and Empty_Text are equality and inequality:

```
if Text = Sentence then
   . . .
end if;
```

Corresponding to this private type declaration there must also be a full type declaration which provides the physical implementation of the type. The full type declaration for Text_Type is the one given at the beginning of the Tutorial. It still appears in the package specification. It is, however, no longer in the public but in the private part of the package specification. Only packages may have a private part, which is that part of the package specification that follows the key word **private**. The structure is as follows:

```
package Text_Package is

    type Text_Type is private;

    function Text_From_String (Words : String) return Text_Type;

    procedure Delete (Phrase : in String;
                      Text   : in out Text_Type);

    procedure Append (To_Text   : in Text_Type;
                      More_Text : in out Text_Type);

    function "<" (Text_1, Text_2 : Text_Type) return Boolean;

    function ">" (Text_1, Text_2 : Text_Type) return Boolean;

    procedure Insert (More_Text    : in String;
                      After_String : in String;
                      To_Text      : in out Text_Type);

    function Is_in_String (Phrase : String; Text : Text_Type)
                          return Boolean;

    function Value (Text : in Text_Type) return String;

private

    subtype Character_Count_Type is Integer range 0 .. 127;

    subtype Text_String is String (1 .. Character_Count_Type'Last);

    type Text_Type is
        record
            Character_Count : Character_Count_Type;
            Text            : Text_String;
        end record;

end Text_Package;
```

The implementation of the private type is not physically hidden from the user of the abstraction -- after all, it is fully declared in the source code of the package specification. The implementation, however, is logically hidden from the user because he is unable to take advantage of it. (The full type declaration of a private type is in the package specification rather than the body to enable the compiler to allocate storage for objects of the type declared by importers of the package.) Consequently, the form in which values for Text_Type must be written is also irrelevant: it could be a record aggregate, an array aggregate, an access value, an enumeration literal, etc. The point is that because the implementation is not available, the user cannot assign a literal to a Text_Type object.

Notice the additional function Value in the above declaration of Text_Package. This subprogram allows the user to retrieve text in the form of a string.

There is a second class of private types, limited types. They include private types declared to be limited private, task types (beyond the scope of this workbook), types derived from a limited type (see Exercise 7.2), and composite types one of whose components is itself limited. Limited types are subject to the same restrictions as private types, with the additional rule that assignment, equality and inequality are not available. These operations are not allowed because there are cases when the result of applying the predefined operation to the object of the private type, given its underlying representation, yields an unanticipated result in the context of the abstraction.

To illustrate this point, consider a more realistic definition of the text abstraction discussed earlier, to allow for text of any length. Suppose that the implementation of text is a linked list of text nodes, such that each node contains a block of text and its character count.

```
package Text_Package is

    type Text_Type is limited private;

    function Text_From_String (Words : String) return Text_Type;

    procedure Delete (Phrase : in String;
                      Text   : in out Text_Type);

    procedure Append (To_Text   : in Text_Type;
                      More_Text : in out Text_Type);

    function "<" (Text_1, Text_2 : Text_Type) return Boolean;

    function "=" (Text_1, Text_2 : Text_Type) return Boolean;

    function ">" (Text_1, Text_2 : Text_Type) return Boolean;

    procedure Insert (More_Text    : in String;
                      After_String : in String;
                      To_Text      : in out Text_Type);

    function Is_in_String (Phrase : String; Text : Text_Type)
                          return Boolean;

    procedure Assign (Some_Text : in Text_Type;
                      To_Text_2 : in out Text_Type);

    function Value (Text : in Text_Type) return String;

private

    subtype Character_Count_Type is Integer range 0 .. 127;

    subtype Text_String is String (1 .. Character_Count_Type'Last);

    type Text_Block_Type is
        record
            Character_Count : Character_Count_Type;
            Text            : Text_String;
        end record;

    type Text_Node_Type;

    type Text_Type is access Text_Node_Type;

    type Text_Node_Type is
        record
            Text_Block : Text_Block_Type;
            Next_Block : Text_Type;
        end record;

end Text_Package;
```

If the predefined operations of assignment and equality were allowed, then the user would find that Text_Type objects exhibit surprising (and undesirable) behavior. Comparing two Text_Type objects using the predefined equality operator results in testing whether or not the Text_Block components and the Next_Block components have the same values. In the latter case, it would check that the access values, as opposed to the accessed values, are the same. Assignment also is problematic. In assigning Sentence to Text, for instance, the pointers to the blocks of text would become identical. The Text object is not copied into the Sentence object, as the user had intended. This inadvertent sharing of objects means that a change to one object would be automatically reflected in the other, a design "feature" sure to irritate the user of the package.

Objects of type Text_Type may be passed as parameters to the procedures and functions defined above. The subprograms declared in Text_Package provide the only ways to manipulate objects of the limited private type Text_Type. The basic operation of assignment is forbidden, and its functionality, if desired, must therefore be explicitly provided, as shown in the package body above. The assignment operator ":=" may *not* be overloaded, and a procedure must be declared to accomplish the same effect. The predefined equality and inequality operators may not be used; however, the package writer is allowed to overload them. Overloading the equality operator "=" implicitly overloads the inequality operator. These operators may only be overloaded for limited private types.

It being illegal to assign string literals to the text, the function Text_From_String converts an arbitrary length string into a Text_Type value. For example, to create text from the string "test sequence loaded" the user uses the Text_From_String function and the Assign procedure:

```
with Text_Package; use Text_Package;
procedure Text_Edit is

    Text_1      : Text_Type;
    Test_Phrase : constant String := "test sequence loaded";

    . . .

begin  -- Text_Edit

    Assign (Some_Text => Text_From_String (Test_Phrase),
            To_Text_2 => Text_1);

    . . .

end Text_Edit;
```

Both the Assign and the Text_From_String subprograms must be used because assignment is not allowed for objects of the limited private type Text_Type. Corresponding to this package specification is a package body in which the subprograms defined in the specification are implemented. Within the package body, the declarations defined in the private part of the specification are visible. In other words, the package body is the only part of the program which can exploit the implementation of the private type. Its purpose is precisely to implement the abstract operations in terms of the concrete implementation of the type.

The subprogram bodies are written no differently than those for subprograms declared in a package specification without a private part. In this case, due to pointer manipulations, the bodies are in fact complex; however, this complexity is transparent to the user. The interface provided by the package specification is clean, and it defines the properties of Text_Type objects. Whether Text_Type is an array of characters, a record, or an access type, to name several alternative implementations, does not affect the result of the various operations. It is important to note that even if the package implementer changes the representation of Text_Type, the user does not need to modify his code in any way. Because of his program's dependence on Text_Package, though, he will have to recompile programs that import Text_Package.

This Tutorial began with a discussion of abstraction, encapsulation and information hiding. How well does Text_Package illustrate these principles? The purpose of an abstraction is to separate the concept from the implementation, enabling the user to discuss the idea at a logical level. Without an abstraction, the salient properties of some entity are lost among the myriad details of its construction, detracting from a program's readability, reusability and maintainability. Suppose for a moment that Text_Package did not exist and that text manipulations were done through in-line code instead. The resulting code would be cluttered and ambiguous in purpose. Adjusting the bounds of slices that hold pieces of text would be a tedious and error-prone process because of the frequency with which this operation is performed. Contrast this situation with the actual Text_Package declared earlier. The user is independent: he is given an abstract type, Text_Type, which he can append to, delete from, search, assign, etc. He manipulates text at the abstract level, and there are no pointer or character count updates to distract the reader or writer of the code.

A good abstraction provides the user with a type and a set of operations on that type, these operations being the interface between the logical entity seen by the user and the physical entity seen by the implementer. The package is the ideal mechanism for making the abstraction available to the user. It encapsulates the type and its operations in a single entity which the user can import where needed. Furthermore, the package designer has complete control over the manipulation of the data, concentrating responsibility for the integrity of the data in a single place. Text_Package exhibits this functionality. The data type Text_Type and the operations on it are physically grouped together in a package because it is their combination that defines the abstraction.

The package Text_Package illustrates information hiding as well through its use of private types. Notice how there are more types declared in the private part of Text_Package than are declared private in the public part of the package. The definition of Text_Type depends on these other types, but they do not belong to the interface between the type and its user. They were therefore omitted from the public part of the specification. The declaration

of Text_Type as limited private reinforces the concept of data abstraction because it consciously separates the user's concerns from those of the implementer. Furthermore, by enclosing the selected linked list representation inside a shatterproof container with well-marked knobs to alter the values inside this box, it guarantees the integrity and consistency of the text data. This valuable guarantee eliminates many errors that might otherwise not surface until much later in the software development life cycle.

There are several technical aspects of private types which the Text_Package example did not show. These are discussed in the remainder of the Tutorial. Private types may have discriminant parts, and the discriminant part belongs to the external interface. The discriminant part is repeated both in the private type and the full type declarations. The selected component operation may be applied to objects of a private or limited private type with a discriminant for the selection of any discriminant component. Thus, given:

```
package Matrix_Package is

    type Square_Matrix_Type (Size : Positive) is private;

    . . .

private

    type Two_Dimensional_Array_Type is array
             (Integer range<>, Integer range<>) of Integer;
    type Square_Matrix_Type (Size : Positive) is
        record
            Matrix : Two_Dimensional_Array_Type
                          (1 .. Size, 1 .. Size);
        end record;

end Matrix_Package;
```

the following is legal:

```
with Matrix_Package; use Matrix_Package;
function Number_of_Elements (Matrix : Square_Matrix_Type)
                              return Positive is
begin
    return Matrix.Size * Matrix.Size;
end Number_of_Elements;
```

The membership operation is legal for limited private types with discriminants. The example below shows how to test the value of the discriminant:

```
if Matrix in Square_Matrix_Type (Size => 10) then
   ...
end if;
```

Constants may be declared of types that are private types. Normally in constant declarations, the initial value of the constant must be specified at the time the constant is declared. Because the implementation of a private type is not known, a constant declared in the visible part cannot be given a value. In Ada, such a constant is known as a deferred constant, and its value must be provided in the private part of the package, as shown below:

```
package Matrix_Package is

    type Square_Matrix_Type (Size : Positive := 2) is private;

    Identity_Matrix_2 : constant Square_Matrix_Type;

    . . .

private

    type Two_Dimensional_Array_Type is array
            (Integer range<>, Integer range<>) of Integer;
    type Square_Matrix_Type (Size : Positive := 2) is
        record
            Matrix : Two_Dimensional_Array_Type
                        (1 .. Size, 1 .. Size);
        end record;
    Identity_Matrix_2 : constant Square_Matrix_Type :=
                        (2, ((1, 0), (0, 1)));

end Matrix_Package;
```

Deferred constants may be declared of private or limited private types. Between the declaration of the deferred constant and its full declaration in the private part of the package, certain rules must be observed. A private type constant may only be used in default expressions for record components or for formal parameters. In the case of a limited private type, however, this constant may not be used in default expressions (i.e. initialization of record components or objects) because the assignment operation is not available for

this class of types. It would seem that there are no uses for limited private constants, but they can be used as the default expressions for formal parameters of mode in subprogram specifications or as actual parameters in subprogram calls.

There are several restrictions to be observed for private types:

- The full type declaration for a private type must not be an unconstrained array type.

- A composite type any of whose components' types is limited private is itself treated as a limited type.

- Catenation is not allowed for array types whose components are of a limited type.

- Explicit initializations of objects (variables and constants) of a limited private type are not allowed outside the package in which the full declaration of the limited private type occurs. Effectively, this rule means that the user of a limited private type (i.e. the programmer who imports a package containing the limited private type LP) cannot declare constants of this type (i.e. of type LP).

- Subprograms with formal parameters of a limited private type of mode out or in out must be declared in the same package specification as the one containing the full declaration of this limited private type. (As an example, in a procedure which imports the package Text_IO, it is illegal to declare a procedure with an out parameter of type File_Type.)

- No initial value may be given for allocated variables of a limited private type.

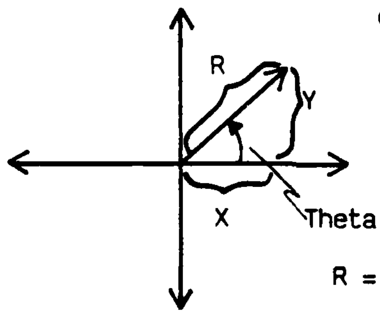
- The attributes 'Size and 'Constrained may be applied to private and limited private types. The attribute 'Size returns the amount of storage in bits. The attribute 'Constrained returns a Boolean value indicating whether the type is constrained.

Problem

An air traffic control system tracks the speed, direction and altitude of planes in its control area. Ignoring the altitude for this problem, assume that the direction and speed are related as follows. Direction is specified by a vector quantity. The magnitude of this vector represents the distance traveled by the aircraft in one minute. Increasing or decreasing the ground speed of the vehicle affects the vector accordingly. Commands from the control tower, issued once a minute, are given in terms of an increase or decrease in speed (i.e. increase speed to 7.2 miles per minute), and a turn of so many degrees, where the number of degrees lies between 0.0 and 360.0. The pilot responds by indicating his new position in terms of his relative motion in the X and Y directions of the Cartesian coordinate system. The control tower computer uses this information to plot the actual flight path and to report to the tower the deviations, if any, from the expected flight path, and the current speed. Assume the plotter expects two inputs, one representing distance traveled in the X direction and the second in the Y direction. (The expected flight path is computed based on the tower's commands to the airplane.)

Write a program for the control tower computer that tracks an airplane's flight path by monitoring the transmissions between the tower and the plane, plots the expected and actual flight data, and reports the status of the flight. Assume that these transmissions are in machine readable form (as opposed to voice transmissions). Ignore timing considerations in the sequencing of the tower and the pilot transmissions. Assume that there is sufficient time delay so that the pilot's position update reflects the command that the tower radioed. The control tower program should refer to the plane's motion in abstract, vector terms. Do not write the code to graph the flight data. Also, do not write code to report the deviations, a subprogram call is sufficient. Assume that the curvature of the earth is not significant.

(The following relationships hold between the Cartesian and the polar coordinate representation:)



for X > 0, Y >= 0, the following equation holds:

$$\text{Theta} = \arctan \frac{Y}{X}$$

Other angles may be derived from the above equation.

$$R = \sqrt{X ** 2 + Y ** 2}$$

Assume the existence of the following math and tower communication packages:

```
package Math_Package is

-- trigonometric functions assume angles are in degrees.

    Pi : constant := 180.0;

    function sin (Degrees : Float) return Float;
    function cos (Degrees : Float) return Float;
    function tan (Degrees : Float) return Float;
    function csc (Degrees : Float) return Float;
    function sec (Degrees : Float) return Float;
    function cot (Degrees : Float) return Float;
    function arcsin (Degrees : Float) return Float;
    function arccos (Degrees : Float) return Float;
    function arctan (Degrees : Float) return Float;
    function Sqrt (X : Float) return Float;
    function Cube_Root (X : Float) return Float;
    function exp (X : Float) return Float;
    function log_10 (X : Float) return Float;
    function ln_e (X : Float) return Float;

end Math_Package;
```

```
package Tower_Communications is

    subtype Distance_Type is Float range -9.9999E10 .. 9.9999E10;
    subtype Speed_Type is Float range 0.0 .. 9000.0;
    subtype Degrees_Type is Float range 0.0.. 360.0;

    procedure Get_Speed (Speed : out Speed_Type);
    procedure Get_Angle (Angle : out Degrees_Type);
    procedure Get_Pilot_Report
                  (X_Position, Y_Position : out Distance_Type);

end Tower_Communications;
```

Solution and Discussion

The required program involves many vector manipulations; it makes sense, therefore, to encapsulate these vector operations in a single package. This package will provide its user, the control tower computer program, with a set of vector tranformations that plot the plane's flight path, compute the deviation from the expected flight path, and so forth. Because a vector can be represented in more than one way (Cartesian coordinates or polar coordinates may be implemented either as a record or an array), a private type is the most appropriate representation of the flight vector. Thus, the control tower applications program can use the vector abstraction independent of its implementation.

```
package Vector_Package is

    type Vector_Type is private;

    type Scalar_Type is digits 5 range -9.9999E10 .. 9.9999E10;

    . . .           -- operations on vectors

private

    type Vector_Type is
       record
           X_Component, Y_Component : Scalar_Type;
       end record;

end Vector_Package;
```

Analyzing the requirements for this library unit reveals a need for the following vector operations:

- addition / subtraction (deviation from flight path)
- rotation (change in direction)
- creation of a vector given its polar representation
- creation of a vector given its Cartesian representation
- X and Y components of a vector in Cartesian form
- angle of a vector
- magnitude of a vector (speed = distance = rate * time, where distance is the magnitude of the vector and time = 1 minute)

The specification of this package can now be written:

```
package Vector_Package is

    type Vector_Type is private;
    type Angle_Type is digits 5 range 0.0 .. 360.0;
    type Scalar_Type is digits 5 range -9.9999E10 .. 9.9999E10;

    function "+" (Left, Right : Vector_Type) return Vector_Type;
    function "-" (Left, Right : Vector_Type) return Vector_Type;
    function Rotation (Vector : Vector_Type;
                       Angle  : Angle_Type)
                      return Vector_Type;

    function Vector_From_Cartesian (X, Y: Scalar_Type)
                                   return Vector_Type;
    function Vector_From_Polar (Magnitude : Scalar_Type;
                                Angle     : Angle_Type)
                               return Vector_Type;
    function X_Component (Vector : Vector_Type) return Scalar_Type;
    function Y_Component (Vector : Vector_Type) return Scalar_Type;
    function Vector_Magnitude (Vector : Vector_Type)
                              return Scalar_Type;
    function Vector_Angle (Vector : Vector_Type) return Angle_Type;

    Zero_Vector : constant Vector_Type;

private

    type Vector_Type is
        record
            X_Component, Y_Component : Scalar_Type;
        end record;

    Zero_Vector : constant Vector_Type := (0.0, 0.0);

end Vector_Package;
```

(Note that a more general approach in defining a vector package would include the operations of scalar multiplication and vector multiplication.)

Notice the introduction of the deferred constant, Zero_Vector. This vector implies no forward motion, thus it can be used to indicate when the plane has reached its destination. The complete code for the package body of Vector_Package is shown at the end of this section. It is noteworthy to mention here the introduction of the procedure Decompose_Angle in the package body:

```
procedure Decompose_Angle
             (Angle                   : in Angle_Type;
              Angle_Sin, Angle_Cos : out Scalar_Type)
             is separate;
```

It is not a part of the vector abstraction and is therefore not included as one of the operations on vectors. It computes the sine and cosine of the angle passed to it as a parameter and as such, it encapsulates an algorithm used both by the Rotation function and by the Vector_from_Polar function. Note also in the package body that the bodies of the overloaded functions "+," "-" and "*" are not made separate. The language disallows the use of operator designators such as "+" or "-" for the names of separately compiled units, be they library units or subunits. The implementation of the body is found in the complete code at the end of this section.

The vector package is in turn used by the control tower computer. The control system monitors both the tower's instructions and the plane's relative position. It acquires these values through the procedures in Tower_Communications package and must convert these to the appropriate vector type. The procedures in the Tower_Communications package use floating point parameters whereas the vector operations are specified in terms of Angle_Type, Scalar_Type and Vector_Type. The conversions of the input from the tower and the pilot are encapsulated in separate functions. For example, the speed input function may be written:

```
function Speed_Input return Scalar_Type is

    Speed : Tower_Communications.Speed_Type;

begin

    Get_Speed (Speed);
    return Scalar_Type (Speed);

end Speed_Input;
```

The tower's instructions are almost in the form of polar coordinates. The new speed specifies the magnitude of the vector. The turn in degrees specifies the angle by which the current vector must be rotated. The tower's command can now easily be converted to the vector needed to plot incrementally the flight's path:

```
Expected_Plane_Motion :=
    Vector_From_Polar (New_Speed,
                       Vector_Angle (Actual_Plane_Motion));
Expected_Plane_Motion :=
    Rotation (Expected_Plane_Motion, Turn_Angle);
```

Thus the new motion of the plane is directly related to its previous direction.

The pilot's position report is converted into vector format by the Pilot_Position_Input function:

```
function Pilot_Position_Input return Vector_Type is

    X, Y : Tower_Communications.Distance_Type;

begin

    Get_Pilot_Position (X, Y);
    return Vector_From_Cartesian (X, Y);

end Pilot_Position_Input;
```

Since the plane's position is in vector form, the deviation between the expected and the actual flight path is computed by subtracting the two vectors:

```
Actual_Plane_Motion - Expected_Plane_Motion
```

The change in speed is computed by subtracting the old speed from the new speed, where the speed is obtained by taking the magnitude of the direction vector. (The magnitude of this vector, the distance traveled by the plane, divided by the rate, 1 minute, gives the speed.)

```
Old_Speed := Vector_Magnitude (Actual_Plane_Motion);
-- process and update Actual_Plane_Motion
Change_in_Speed :=
    Vector_Magnitude (Actual_Plane_Motion) - Old_Speed;
```

Both the plotting and reporting requirements are implemented as procedures. Because the code for these subprograms is not part of the exercise, they are written as subunits:

```
procedure Plot (X, Y : in Scalar_Type) is separate;

procedure Report_to_Tower
              (Position_Deviation : in Vector_Type;
               Speed_Change       : in Scalar_Type)
              is separate;
```

Because the plot procedure expects the X and Y components of the motion vector, it is called as:

```
Plot (X_Component (Actual_Plane_Motion),
      Y_Component (Actual_Plane_Motion));
```

Certain assumptions are made in coding this procedure. Once the plane begins taxiing for take-off, it does not cease forward motion until it has arrived at the destination gate. Thus, if the vector describing the plane's option gets the value of the zero vector, the plane has reached its destination.

The complete solution code follows.

```
package Vector_Package is

    type Vector_Type is private;
    type Angle_Type is digits 5 range 0.0 .. 360.0;
    type Scalar_Type is digits 5 range -9.9999E10 .. 9.9999E10;

    function "+" (Left, Right : Vector_Type) return Vector_Type;
    function "-" (Left, Right : Vector_Type) return Vector_Type;
    function Rotation (Vector : Vector_Type;
                       Angle  : Angle_Type)
                      return Vector_Type;

    function Vector_From_Cartesian (X, Y: Scalar_Type)
                                    return Vector_Type;
    function Vector_From_Polar (Magnitude : Scalar_Type;
                                Angle     : Angle_Type)
                               return Vector_Type;
    function X_Component (Vector : Vector_Type) return Scalar_Type;
    function Y_Component (Vector : Vector_Type) return Scalar_Type;
    function Vector_Magnitude (Vector : Vector_Type)
                              return Scalar_Type;
    function Vector_Angle (Vector : Vector_Type) return Angle_Type;

    Zero_Vector : constant Vector_Type;

private

    type Vector_Type is
        record
            X_Component, Y_Component : Scalar_Type;
        end record;

    Zero_Vector : constant Vector_Type := (0.0, 0.0);

end Vector_Package;
```

```
package Math_Package is

-- trigonometric functions assume angles are in degrees.

    Pi : constant := 180.0;

    function sin (Degrees : Float) return Float;
    function cos (Degrees : Float) return Float;
    function tan (Degrees : Float) return Float;
    function csc (Degrees : Float) return Float;
    function sec (Degrees : Float) return Float;
    function cot (Degrees : Float) return Float;
    function arcsin (Degrees : Float) return Float;
    function arccos (Degrees : Float) return Float;
    function arctan (Degrees : Float) return Float;
    function Sqrt (X : Float) return Float;
    function Cube_Root (X : Float) return Float;
    function exp (X : Float) return Float;
    function log_10 (X : Float) return Float;
    function ln_e (X : Float) return Float;

end Math_Package;
```

```
package Tower_Communications is

    subtype Distance_Type is Float range -9.9999E10 .. 9.9999E10;
    subtype Speed_Type is Float range 0.0 .. 9000.0;
    subtype Degrees_Type is Float range 0.0.. 360.0;

    procedure Get_Speed (Speed : out Speed_Type);
    procedure Get_Angle (Angle : out Degrees_Type);
    procedure Get_Pilot_Report
                 (X_Position, Y_Position : out Distance_Type);

end Tower_Communications;
```

```
with Tower_Communications; use Tower_Communications;
with Vector_Package; use Vector_Package;
procedure Track_Flight is

    Turn_Angle                   : Angle_Type;

    Change_in_Speed, Old_Speed,
    New_Speed, X, Y              : Scalar_Type;

    Expected_Plane_Motion,
    Actual_Plane_Motion          : Vector_Type;

    function Speed_Input return Scalar_Type is separate;

    function Angle_Input return Angle_Type is separate;

    function Pilot_Position_Input return Vector_Type is separate;

    procedure Plot (X, Y : in Scalar_Type) is separate;

    procedure Report_to_Tower
                  (Position_Deviation : in Vector_Type;
                   Speed_Change       : in Scalar_Type)
                  is separate;

begin  -- Track_Flight

    -- get initial data as plane leaves departure gate

    New_Speed := Speed_Input;
    Turn_Angle := Angle_Input;
    Expected_Plane_Motion := Vector_From_Polar
                                   (New_Speed, Turn_Angle);
    Actual_Plane_Motion := Expected_Plane_Motion;

    -- start plotting flight path

    Plot (X_Component (Expected_Plane_Motion),
          Y_Component (Expected_Plane_Motion));
    Plot (X_Component (Actual_Plane_Motion),
          Y_Component (Actual_Plane_Motion));
```

```
    while Actual_Plane_Motion /= Zero_Vector loop

        New_Speed := Speed_Input;
        Turn_Angle := Angle_Input;
        Expected_Plane_Motion :=
            Vector_From_Polar (New_Speed,
                               Vector_Angle (Actual_Plane_Motion));
        Expected_Plane_Motion :=
            Rotation (Expected_Plane_Motion, Turn_Angle);
        Plot (X_Component (Expected_Plane_Motion),
              Y_Component (Expected_Plane_Motion));
        Old_Speed := Vector_Magnitude (Actual_Plane_Motion);
        Actual_Plane_Motion := Pilot_Position_Input;
        Change_in_Speed :=
            Vector_Magnitude (Actual_Plane_Motion) - Old_Speed;
        Plot (X_Component (Actual_Plane_Motion),
              Y_Component (Actual_Plane_Motion));
        Report_to_Tower (Actual_Plane_Motion - Expected_Plane_Motion,
                         Change_in_Speed);

    end loop;

end Track_Flight;

------------------------------------------------------------------

separate (Track_Flight)
function Speed_Input return Scalar_Type is

    Speed : Tower_Communications.Speed_Type;

begin

    Get_Speed (Speed);
    return Scalar_Type (Speed);

end Speed_Input;

------------------------------------------------------------------

separate (Track_Flight)
function Angle_Input return Angle_Type is

    Degrees : Tower_Communications.Degrees_Type;

begin

    Get_Angle (Degrees);
    return Angle_Type (Degrees);

end Angle_Input;

------------------------------------------------------------------
```

```
separate (Track_Flight)
function Pilot_Position_Input return Vector_Type is

    X, Y : Tower_Communications.Distance_Type;

begin

    Get_Pilot_Report (X, Y);
    return Vector_From_Cartesian (Scalar_Type (X),
                                  Scalar_Type (Y));

end Pilot_Position_Input;
```

```
package body Vector_Package is

    procedure Decompose_Angle
                  (Angle                   : in Angle_Type;
                   Angle_Sin, Angle_Cos : out Scalar_Type)
                              is separate;

    function Rotation (Vector : Vector_Type;
                       Angle  : Angle_Type)
                      return Vector_Type is separate;

    function Vector_From_Cartesian (X, Y: Scalar_Type)
                                   return Vector_Type
                                   is separate;

    function Vector_From_Polar (Magnitude : Scalar_Type;
                                Angle     : Angle_Type)
                               return Vector_Type is separate;

    function X_Component (Vector : Vector_Type) return Scalar_Type
                         is separate;

    function Y_Component (Vector : Vector_Type) return Scalar_Type
                         is separate;

    function Vector_Magnitude (Vector : Vector_Type)
                              return Scalar_Type is separate;

    function Vector_Angle (Vector : Vector_Type) return Angle_Type
                          is separate;

    function "+" (Left, Right : Vector_Type) return Vector_Type is
    begin

        return
            (X_Component =  Left.X_Component + Right.X_Component,
             Y_Component =  Left.Y_Component + Right.Y_Component);

    end "+";

    function "-" (Left, Right : Vector_Type) return Vector_Type is
    begin

        return
            (X_Component => Left.X_Component - Right.X_Component,
             Y_Component => Left.Y_Component - Right.Y_Component);

    end "-";

end Vector_Package;   -- package body
-------------------------------------------------------------------
```

```
with Math_Package; use Math_Package;
separate (Vector_Package)
procedure Decompose_Angle
              (Angle                : in Angle_Type;
               Angle_Sin, Angle_Cos : out Scalar_Type) is

begin  -- Decompose_Angle

    Angle_Sin := Scalar_Type (Sin (Float (Angle)));
    Angle_Cos := Scalar_Type (Cos (Float (Angle)));

end Decompose_Angle;
```

```
separate (Vector_Package)
function Rotation (Vector : Vector_Type;
                   Angle  : Angle_Type)
                  return Vector_Type is

    Angle_Sin, Angle_Cos : Scalar_Type;

begin

    Decompose_Angle (Angle, Angle_Sin, Angle_Cos);
    return
        (X_Component => Vector.X_Component * Angle_Cos -
                        Vector.Y_Component * Angle_Sin,
         Y_Component => Vector.Y_Component * Angle_Cos -
                        Vector.X_Component * Angle_Sin);

end Rotation;
```

```
separate (Vector_Package)
function Vector_From_Cartesian (X, Y: Scalar_Type)
                                   return Vector_Type is
begin

    return (X_Component => X, Y_Component => Y);

end Vector_From_Cartesian;

-------------------------------------------------------------------

separate (Vector_Package)
function Vector_From_Polar (Magnitude : Scalar_Type;
                            Angle     : Angle_Type)
                           return Vector_Type is

    Angle_Sin, Angle_Cos : Scalar_Type;

begin  -- Vector_From_Polar

    Decompose_Angle (Angle, Angle_Sin, Angle_Cos);
    return
        (X_Component => Angle_Sin * Magnitude,
         Y_Component => Angle_Cos * Magnitude);

end Vector_From_Polar;

-------------------------------------------------------------------

separate (Vector_Package)
function X_Component (Vector : Vector_Type) return Scalar_Type is
begin

    return Vector.X_Component;

end X_Component;

-------------------------------------------------------------------

separate (Vector_Package)
function Y_Component (Vector : Vector_Type) return Scalar_Type is
begin

    return Vector.Y_Component;

end Y_Component;

-------------------------------------------------------------------
```

```
with Math_Package; use Math_Package;
separate (Vector_Package)
function Vector_Magnitude (Vector : Vector_Type)
                                  return Scalar_Type is
begin

    return (Scalar_Type
        (Sqrt (Float
            (Vector.X_Component ** 2 + Vector.Y_Component ** 2))));

end Vector_Magnitude;
```

```
with Math_Package; use Math_Package;
separate (Vector_Package)
function Vector_Angle (Vector : Vector_Type) return Angle_Type is

    Principal_Angle : Angle_Type;

begin  -- Vector_Angle

    if Vector.Y_Component = 0.0 then
        if Vector.X_Component >= 0.0 then
            return Pi / 2.0;
        else
            return Pi * 1.5;
        end if;
    else
        Principal_Angle :=
            Angle_Type (arctan (Float (Vector.X_Component) /
                                Float (Vector.Y_Component)));
        if Vector.Y_Component < 0.0 then
            return Principal_Angle + Pi;
        else
            return Principal_Angle;
        end if;
    end if;

end Vector_Angle;
```

EXERCISE 3.3

GENERICS AND OVERLOADING

Objective

This exercise introduces generic units and provides examples of type, object, and subprogram formal parameters.

Tutorial

Generic subprograms and packages are used when there exist similar processes whose differences are not central to their algorithms. For example, a single generic subprogram would be suitable for an operation that is performed on several types of objects. Consider the implementation of a function that, when applied to a discrete type, returns 'Succ unless the given value is last in the type, in which case the function returns the first element. The function for objects of type Month_Type would be:

```
type Month_Type is (January, February, March, April,
                    May, June, July, August, September,
                    October, November, December);

function Next_Month (X : Month_Type) return Month_Type is
begin
    if X = Month_Type'Last then
        return Month_Type'First;
    else
        return X'Succ;
    end if;
end Next_Month;
```

Without generic units, it would be necessary to write a distinct function for every type to which this operation is applied. In other words, if a program contains definitions for several integer and enumeration types, such as,

```
type Locations_Type is (Karachi, Pasni, Rawalpindi,
                        Peshwar, Quetta, Lahore);
type Message_ID_Type is range 1000 .. 10_000;
type Reconnaissance_Range_Type is range 30 .. 400;
```

and if it is expedient to apply the successor operation described above to all of these types, a separate function must be written for each parameter type.

Alternatively, one could write a single generic function:

```
generic
    type Discrete_Type is (<>);      -- Formal type dec.
function Next (X : Discrete_Type) return Discrete_Type;

function Next (X : Discrete_Type) return Discrete_Type is
begin
    if X = Discrete_Type'Last then
        return Discrete_Type'First;
    else
        return Discrete_Type'Succ;
    end if;
end Next;
```

The generic function declaration does not create an actual function, but rather a template for a function with the parameter type left blank. Similarly, a generic package is not a package, but a template from which several similar packages can be "instantiated"; the entities declared within it may only be referenced after an instance of the package has been created, a process called instantiation.

Generic units have both a specification and a body. In generic subprograms the specification consists of the word "generic", the declarations of the generic formal parameters, and the specification of the subprogram. The generic body is the ordinary body of the subprogram. In generic packages, the package declaration follows immediately after the generic formal parameters (explained below). The generic formal parameter of the function above is Discrete_Type. Generic parameters may represent (1) types, (2) constants, (3) variables, and (4) subprograms called from within the template. The inclusion of parameters in the definition of a generic unit is not required, but they are usually given. Generic units that have no parameters are not very useful because all instances of them are identical.

The declaration of Discrete_Type (characterized by the symbol "(<>)") in the generic function Next indicates that the types of parameters with which Next may be instantiated include only discrete types (i.e. integer and enumeration). Instances of Next may be created for each of the discrete types mentioned earlier as follows:

```
function Next_Month is new Next (Month_Type);
function Next_Location is new Next (Locations_Type);
function Next_Message_ID is
      new Next (Message_ID_Type);
function Next_Reconnaissance_Range is
      new Next (Reconnaissance_Range_Type);
```

In these four instantiations the types Month_Type, Locations_Type, Message_ID_Type, and Reconnaissance_Range_Type are actual parameters that correspond to the formal parameter in Next. These instantiations are completely equivalent to the declaration of four functions, with the respective types replacing Discrete_Type in Next. For example, the first instantiation is equivalent to the declaration of Next_Month, defined earlier, and the second instantiation is equivalent to the declaration of the function,

```
function Next_Location (X : Locations_Type)
                       return Locations_Type is
begin
    if X = Locations_Type'Last then
        return Locations_Type'First;
    else
        return X'Succ;
    end if;
end Next_Location;
```

Notice that the name of the instance of the function is given after the word "function," or, as the case may be, "procedure" or "package." It is perfectly legal to instantiate a generic unit several times with the same name (as long as the types are different), in which case that name is said to be "overloaded." Consider the following example.

It is useful in a system that processes aerial photographic data to define a function that, given the density (level of darkness) of the image over a square millimeter, returns a Boolean value that corresponds to black or white. The density of the photographs taken by different cameras are declared floating point types. In general, the more sophisticated models have a density represented by many digits of accuracy, whereas simpler models have only a few significant digits. Also, the density is measured on different scales, depending upon the film that the camera uses. Three density types are given below. Others will later be added to the system.

```
type Average_Density_Model_1_Type is
                digits 3 range -1.0 .. 1.0;
type Average_Density_Model_2_Type is
                digits 5 range -25.0 .. 25.0;
type Average_Density_Model_3_Type is
                digits 7 range 0.0 .. 10.0;
```

The function that processes the density is to be symbolized by the plus sign, "+"; it returns true if the average density is in the positive half, or upper half, of the range for that type, and false if it is in the lower half. Because the function must accept different density types, a generic function is written. One could declare all of the density types as subtypes of a base type which would serve as a parameter type. However, in this case, one does not know the specifications of the density types that would later be added to the system.

```
generic
    type Parameter_FP_Type is digits <>;  -- Formal type dec.
function Density (X : Parameter_FP_Type)
                return Boolean;

function Density (X : Parameter_FP_Type)
                return Boolean is
    Halfway_Value : constant Parameter_FP_Type :=
            (Parameter_FP_Type'Last -
             Parameter_FP_Type'First) / 2.0;
begin
    return X >= Halfway_Value;
end Density;
```

The declaration of the formal generic parameter Parameter_FP_Type stipulates that Density may be instantiated with any floating point type. A "+" is not given as the name of the function because, unlike ordinary functions, the name of a generic function must be an identifier.

Instances of the generic function are created by the following declarations:

```
function "+" is new Density (Average_Density_Model_1_Type);
function "+" is new Density (Average_Density_Model_2_Type);
function "+" is new Density (Average_Density_Model_3_Type);
```

The symbol "+" is then overloaded. In fact, the plus sign is overloaded in two senses. It is overloaded by the first instantiation because the plus sign is defined in the package Standard. Additionally, it is overloaded because there are three instantiations of the same sign. An instantiation may overload another instantiation, an ordinary subprogram, or a predefined subprogram (as long as the types are different).

There is no ambiguity resulting from the instantiations above. When an instance of the function is called, the function is determined by the type of the parameter given in the call. It would be illegal, however, (and senseless) to instantiate Density twice with the same name and the same type given for Parameter_FP_Type.

In a generic unit there can be three kinds of formal parameters: objects (for values and variables), subprograms (for procedures and functions with specified parameter and result types), and types (for types or subtypes). So far, only the forms of generic formal parameter types for discrete and floating point types have been seen. There are several other allowed types for generic formal parameters (discrete and floating point are included for the sake of completeness):

Generic Formal Types

```
type Identifier is (<>);                    -- Discrete type
type Identifier is range<>;                 -- Integer type
type Identifier is digits<>;                -- Floating point type
type Identifier is delta<>;                 -- Fixed point type
type Identifier [Discriminant_Part] is
        [limited] private;                  -- Private type
type Identifier is
        array (Index_Subtype, {Index_Subtype})
                of Component_Type;          -- Array type
type Identifier is access Designated_Subtype; -- Access type
```

These declarations look a little like ordinary type declarations, but they are specifically for parameters in generic units. Notice that there is no generic formal record type.

Within a generic unit, a formal type parameter stands for a class of types which have only the operations associated with that class. In a generic instantiation, the corresponding actual parameter may be of any type or subtype as long as its class has all of the operations that are available to the formal parameter type. For example, in the body of the following generic procedure,

```
generic
    type Parameter_Type is private;       -- Formal type dec.
procedure Exchange (X,Y : Parameter_Type);
```

only "=", "/=", ":=", and user-defined subprogram calls are available for the type Parameter_Type. For example, the expression,

```
X + Y
```

inside the body of Exchange would be illegal. In an instantiation of this procedure any non-limited type may be given as the actual parameter. In general terms, the formal parameter types that enable more versatility within the body of the generic unit, incur more restrictions on the type of the corresponding actual parameter. Conversely, a formal parameter type that allows few operations within the generic unit, may be instantiated with many different types.

Whereas a generic formal type is used as a type within the generic unit, a generic formal object stands for either a constant or a variable. Formal objects are used, in a generic unit, exactly as ordinary constants and variables are. Consider the following example.

```
generic
    Multiple : in Integer;               -- Formal constant dec.
procedure Fixed_Multiply (X : in out Integer);

procedure Fixed_Multiply (X : in out Integer) is
begin
    X := X * Multiple;
end Fixed_Multiply;
```

An instance of this procedure multiplies the parameter X by a number determined in the instantiation. For example,

```
procedure Multiply_by_5 is
        new Fixed_Multiply (Multiple => 5);
```

creates a procedure named Multiply_by_5, which multiplies a number by 5. Multiple, declared in Fixed_Multiply, is a generic formal constant. Multiply_by_5 will always deliver the product of the parameter and 5. Notice also, that in an instantiation, the parameter may be given using named notation.

In the generic body above, the generic formal object is a constant and stands for the actual value given in the instantiation, and it may be used like an ordinary constant. A formal constant takes the form:

```
Identifier {,Identifier} : [in] Type_or_Subtype_Name;
```

Formal constants may not be of mode out. (The word "in" may be omitted, but this practice is not recommended.)

A generic formal variable, on the other hand, stands for an actual variable, and is used within the generic body as an ordinary variable. It takes the form:

```
Identifier {,Identifier} : in out Type_or_Subtype_Name;
```

An example of a generic formal variable follows:

```
generic
    Multiple : in out Integer;          -- Formal variable dec.
procedure Variable_Multiply (X : in out Integer);

procedure Variable_Multiply (X : in out Integer) is
begin
    X := X * Multiple;
end Variable_Multiply;
```

This procedure is almost the same as the preceding one. Like Fixed_Multiply, Variable_Multiply also multiplies a number by the value of the actual parameter in the instantiation of the procedure. The difference is that the actual parameter in Variable_Multiply must be a variable (possibly an array component, record component, or allocated variable). Thus, given two variables P and Q,

```
P : Integer := 4;
Q : Integer := 1;
```

instantiations of the generic procedures above,

```
Multiply_by_4 is new Fixed_Multiply (P);
Multiply_by_P is new Variable_Multiply (P);
```

create procedures that multiply by some value. The first, Multiply_by_4, will always multiply by the value of the actual parameter at the time of instantiation, that is by four; the second, Multiply_by_P, will multiply by the value of the variable P at the time the procedure is called. Note that Q is quadrupled by the calls:

```
Multiply_by_4 (Q);          -- Q becomes 4.
Multiply_by_P (Q);          -- Q becomes 16.
```

However, if P is changed to ten,

```
P := 10;

Multiply_by_4 (Q);          -- Q becomes 64.
Multiply_by_P (Q);          -- Q becomes 640.
```

Multiply_by_P will now multiply a number by ten, whereas Multiply_by_4 will always multiply a number by four.

Generic packages are templates from which ordinary packages can be defined by instantiation. The following example involves a group of functions and type declarations, which are incorporated into a generic package.

The set operation for the union of two sets is symbolized by a "+". (Union, as you recall from Exercise 2.1, delivers a set that contains every member of both sets with no duplicates.) The operation for intersection is symbolized by a "*" (common elements), and the operation for set difference is symbolized by a "-" (non-common elements). If a set of letters is defined,

```
type Alphabet_Type is (A, B, C, D, E, F, G, H, I,
                       J, K, L, M, N, O, P, Q, R,
                       S, T, U, V, W, X, Y, Z);
type Set_Type is array (Alphabet_Type) of Boolean;
```

the function for union is written:

```
function "+" (Left, Right : Set_Type)
          return Set_Type is
begin
   return Left or Right;
end "+";
```

The disadvantage of using an ordinary function is that it may only be applied to a set of the defined type Set_Type. By using a generic package we can define once and for all a general purpose set capability. The generic package is written:

```
generic
    type Element_Type is (<>);         -- Formal discrete type.
package Set_Package is

    type Set_Type is private;

    type Element_List_Type is
            array (Positive range <>) of Element_Type;

    function Set_Creation
            (Element_List : Element_List_Type)
            return Set_Type;

    function "+" (Left, Right : Set_Type)      -- Union.
                 return Set_Type;

    function "*" (Left, Right : Set_Type)      -- Intersection.
                 return Set_Type;

    function "-" (Left, Right : Set_Type)      -- Difference.
                 return Set_Type;

private
    type Set_Type is
            array (Element_Type) of Boolean;   -- Set type.

end Set_Package;

package body Set_Package is

    function Set_Creation
            (Element_List : Element_List_Type)
            return Set_Type is
        Set : Set_Type := (Set_Type => False);
    begin
        for I in Element_List'Range loop
            Set(Element_List(I)) := True;
        end loop;
    end Set_Creation;
```

```
        function "+" (Left, Right : Set_Type)
                    return Set_Type is
        begin
            return Left or Right;
        end "+";

        function "*" (Left, Right : Set_Type)
                    return Set_Tvpe is
        begin
            return Left and Right;
        end "*";

        function "-" (Left, Right : Set_Type)
                    return Set_Type is
        begin
            return Left and not Right;
        end "-";

    end Set_Package;
```

The declaration of the set type is private because it is not necessary for the user to know the specific implementation of it.

If the Set_Package is instantiated with the letter type declared previously,

```
Alphabet_Set_Package is new Set_Package (Alphabet_Type);
```

and "use"ed,

```
use Alphabet_Set_Package;
```

the type Set_Type and the three functions declared in the package become available. The functions are available for sets having elements of Alphabet_Type. One can declare sets,

```
Set_A : Set_Type :=
        (Set_Creation ((A, B, C, D, E, F, G, H)));

Set_B : Set_Type :=
        (Set_Creation ((A, B, C, D, E, F, G, H,
                        I, J, K, L, M, N, O, P)));

Set_C : Set_Type;
```

and invoke the set functions:

```
Set_C := Set_A + Set_B;
Set_C := Set_A * Set_B;
Set_C := Set_A - Set_B;
```

Naturally, Set_Package may be instantiated with several different types, and the set functions can be applied to them without ambiguity.

Up to this point, the generic formal parameters discussed have been for types and objects. One can also use generic formal subprograms which stand for actual procedures or functions that have the same parameter and/or result types. The format for a generic formal subprogram declaration is:

```
with Subprogram_Specification;
```

A generic formal subprogram could be used, for example, to apply a given function or procedure to individual elements of an array. Given the declarations,

```
type Real is digits 4 range -500.0 .. 500.0;
type Real_Array_Type is
        array (1 .. 10) of Real;
type Integer_Array_Type is
        array (1 .. 12) of Integer;
function Truncated_at_Zero (X : Real) return Real is
begin
    if X < 0.0 then
        return 0.0;
    else
        return X;
end Truncated_at_Zero;

function Increment (X : Integer) return Integer is
begin
    return X + 1;
end Increment;
```

the generic procedure,

```
generic
    type Element_Type is private;
    type Array_Type is
            array (Positive range <>) of Element_Type;
    with procedure Process_One_Element
            (X : in out Element_Type);
procedure Process_Each_Element (Y : in out Array_Type);

procedure Process_Each_Element (Y : in out Array_Type) is
begin
    for I in Array_Type'Range
    loop
        Process_One_Element (Y(I));
    end loop;
end Process_Each_Element;
```

can be instantiated with those functions, as in,

```
procedure Remove_Negative_Numbers is
      new Process_Each_Element
            (Real, Real_Array_Type, Truncated_at_Zero);
procedure Increment_Array_Elements is
      new Process_Each_Element
            (Integer, Integer_Array_Type, Increment);
```

Given these object declarations,

```
Numbers : Integer_Array_Type :=
      (1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144);

Static_Evaluations : Real_Array_Type :=
      (1.0, 5.1, -12.1, 3.4, 2.1,
      -7.0, -8.0, 1.2, 1.0, -11.0);
```

these procedures may be called with the following effects:

```
Remove_Negative_Numbers           -- Static_Evaluations will look like:
      (Static_Evaluations);       -- (1.0, 5.1, 0.0, 3.4, 2.1,
                                  --  0.0, 0.0, 1.2, 1.0, 0.0)

Increment_Array_Elements          -- Numbers will look like:
      (Numbers);                  -- (2, 2, 3, 4, 6, 9, 14,
                                  --  22, 35, 56, 90, 145)
```

Generic subprogram parameters can also be used to access functions that are normally unavailable to the objects of the formal parameter type. For example, if a generic formal parameter is of type **limited private**, as in,

```
generic
    type Element_Type is limited private;
function Transfer (Y : Element_Type) return Element_Type;
```

no operations are allowed in the function body, except subprogram calls. If equality and inequality are needed, they can be made available by using a generic subprogram parameter, as follows,

```
generic
    type Element_Type is limited private;
    with function Equal (X, Y : Element_Type) return Boolean;
    with function Not_Equal (X, Y : Element_Type) return Boolean;
function Transfer (Y : Element_Type) return Element_Type;
```

If Transfer is instantiated with "=" and "/=", Equal and Not_Equal are used in the body of the function exactly as "=" and "/=" would be.

Lastly, generic units, like ordinary subprograms and packages, may be declared where other declarations occur. If the specification of a generic unit is in the declarative region of a package or subprogram, the generic body must be in the corresponding body.

Problem

A fruit-growing company named Elod has locations in Hawaii, Mexico, and the Virgin Islands. At each location there are three fruits grown, grapefruit, limes, and pineapples, and ten trees of each fruit. Each tree produces between zero and twenty-five bushels a year. The inventory must be analyzed to find many values: the total yield for each fruit, the total yield for each location, the number of trees yielding more than a given number of bushels, and the number of a given type of tree yielding more than a given number of bushels.

Write a package that declares the necessary types and functions to make the calculations outlined above, and reads the inventory information from an external file. Then write a procedure that uses that package to write a report containing the following information:

1. The total annual yield in bushels of the entire company.
2. The total annual yield in bushels of each location.
3. The total annual yield in bushels of each fruit.
4. The total annual yield in bushels of grapefruit grown in Mexico.
5. The total number of trees producing fewer than five bushels.
6. The total number of lime trees producing more than fifteen bushels.
7. The total number of trees in the Virgin Islands that produce exactly twenty bushels.

Solution and Discussion

First the basic types needed can be written for the fruit, location, tree number, bushels, and inventory:

```
type Location_Type is (Hawaii, Mexico, Virgin_Islands);

type Fruit_Type is (Grapefruits, Pineapples, Limes);

type Tree_Number_Type is range 1 .. 10;

subtype Bushels_Type is Integer range 0 .. 25;

type Inventory_Type is array (Location_Type,
                              Fruit_Type,
                              Tree_Number_Type) of Bushels_Type;

Inventory : Inventory_Type;
```

Tree_Number_Type could also be an enumeration type with enumeration literals T1, T2, etc. because it is not used for numeric calculations. But since tree numbers are just that, numbers, it is just as good as an integer type. Bushels_Type is a subtype of Integer so that type mismatches will not occur during numeric calculations. Inventory_Type could also have been an array of a two-dimensional array, or an array of an array of an array. There are advantages, pertaining to the flexibility of accessing, to each structure. One cannot, for example, reference a slice of a two- or three-dimensional array. In this problem, however, the array is searched only by using a nested **for loop**, which lends itself to the simplest structure, a three-dimensional array.

Now to turn to the required functions. There are two types of functions outlined in the problem. They are those that count the number of bushels grown on certain trees, and those that count the number of trees that yield a certain number of bushels. So we will strive for two generic functions, Sum_in_Bushels and Sum_of_Trees. Both will have generic formal parameters corresponding to Location_Type, and Fruit_Type. The third index of the inventory array Tree_Number_Type does not change, so there is no need for a third generic parameter. Thus, Sum_in_Bushels looks like:

```
generic
    type Index1_Type is (<>);          -- Location.
    type Index2_Type is (<>);          -- Fruit.

function Sum_in_Bushels return Integer;

function Sum_in_Bushels return Integer is

    Bushels : Integer := 0;

begin    -- Sum_in_Bushels.

    for I in Index1_Type loop
        for J in Index2_Type loop
            for K in Tree_Number_Type loop
                Bushels := Bushels + Inventory(I,J,K);
            end loop;
        end loop;
    end loop;

    return Bushels;

end Sum_in_Bushels;
```

Sum_of_Trees will additionally have a generic parameter for the function that determines the trees to be included in the count. (The actual parameters corresponding to that function could be "<", ">", or "="). There must also be a parameter in the function to denote the number of bushels in the comparison. So the Sum_of_Trees appears:

```
generic
    type Index1_Type is (<>);          -- Location.
    type Index2_Type is (<>);          -- Fruit.
    with function Comparison
            (A, B : in Bushels_Type) return Boolean;

function Sum_of_Trees (N : Bushels_Type) return Integer;
```

```
function Sum_of_Trees (N : Bushels_Type) return Integer is

    Trees : Integer := 0;

begin     -- Sum_of_Trees.

    for I in Index1_Type loop
        for J in Index2_Type loop
            for K in Tree_Number_Type loop
                if Comparison (Inventory(I,J,K), N) then
                    Trees := Trees + 1;
                end if;
            end loop;
        end loop;
    end loop;

    return Trees;

end Sum_of_Trees;
```

The specifications of Sum_in_Bushels and Sum_of_Trees are declared in the specification of the package, and their bodies are in the package body. The package body also contains the I/O to read in the inventory information from an external file. This code is straightforward, and is essentially,

```
        package Inventory_IO is new Integer_IO (Bushels_Type);
        use Inventory_IO;

        Inventory_File : File_Type;
        Inventory_Name : constant String := "Inventory.Dat";

    begin

        Open (Inventory_File, In_File, Inventory_Name);

        for Location in Location_Type loop
            for Fruit in Fruit_Type loop
                for Tree_Number in Tree_Number_Type loop
                    Inventory_IO.Get (Inventory_File,
                                      Inventory(Location,
                                                Fruit,
                                                Tree_Number));
                end loop;
            end loop;
        end loop;

        Close (Inventory_File);

    end;
```

Now the procedure that makes the report can be written. It contains the instantiations of Sum_in_Bushels and Sum_of_Trees needed for the calculations specified in the problem. The total yield of the company is calculated from the addition of the individual outputs of each location. Objects for those values are declared:

```
Total_Bushels_in_Hawaii  : Integer;
Total_Bushels_in_Mexico  : Integer;
Total_Bushels_in_VI      : Integer;
Total_Bushels_of_Company : Integer;
```

The instantiation of Sum_in_Bushels to determine the annual yield of grapefruit is:

```
function Total_Grapefruits is
        new Sum_in_Bushels
                (Location_Type,
                 Fruit_Type range Grapefruits .. Grapefruits);
```

Total_Pineapples and *Total_Limes* are similarly declared.

The function that calculates the quantity of grapefruits grown in Mexico is instantiated as follows:

```
function Total_Grapefruits_in_Mexico is
        new Sum_in_Bushels
                (Location_Type range Mexico .. Mexico,
                 Fruit_Type range Grapefruits .. Grapefruits);
```

The function that calculates the number of trees producing fewer than five bushels is instantiated with the "<" operator, as follows,

```
function Trees_Yielding_Less_Than is
        new Sum_of_Trees
                (Location_Type,
                 Fruit_Type,
                 "<");
```

The remaining functions are similar.

The report is written to the default output file. A title, starting at the tenth column, and the annual yield information are written.

The entire solution follows:

```
package Elod_Co is

     type Location_Type is (Hawaii, Mexico, Virgin_Islands);

     type Fruit_Type is (Grapefruits, Pineapples, Limes);

     type Tree_Number_Type is range 1 .. 10;

     subtype Bushels_Type is Integer range 0 .. 25;

     type Inventory_Type is
             array (Location_Type,
                    Fruit_Type,
                    Tree_Number_Type) of Bushels_Type;

     Inventory : Inventory_Type;


     generic
         type Index1_Type is (<>);          -- Location.
         type Index2_Type is (<>);          -- Fruit.

     function Sum_in_Bushels return Integer;


     generic
         type Index1_Type is (<>);          -- Location.
         type Index2_Type is (<>);          -- Fruit.
         with function Comparison
                 (A, B : in Bushels_Type) return Boolean;

     function Sum_of_Trees (N : Bushels_Type) return Integer;

end Elod_Co;
```

```
with Text_IO; use Text_IO;
package body Elod_Co is

    function Sum_in_Bushels return Integer is

        Bushels : Integer := 0;

    begin    -- Sum_in_Bushels.

        for I in Index1_Type loop
            for J in Index2_Type loop
                for Tree in Tree_Number_Type loop
                    Bushels := Bushels + Inventory(I, J, Tree);
                end loop;
            end loop;
        end loop;

        return Bushels;

    end Sum_in_Bushels;


    function Sum_of_Trees (N : Bushels_Type) return Integer is

        Trees : Integer := 0;

    begin    -- Sum_of_Trees.

        for I in Index1_Type loop
            for J in Index2_Type loop
                for Tree in Tree_Number_Type loop
                    if Comparison (Inventory(I, J, Tree), N) then
                        Trees := Trees + 1;
                    end if;
                end loop;
            end loop;
        end loop;

        return Trees;

    end Sum_of_Trees;

    package Inventory_IO is new Integer_IO (Bushels_Type);
    use Inventory_IO;

    Inventory_File : File_Type;
    Inventory_Name : constant String := "Inventory.Dat";
```

```
begin    -- Elod_Co.

    Open (Inventory_File, In_File, Inventory_Name);
    for Location in Location_Type loop
        for Fruit in Fruit_Type loop
            for Tree_Number in Tree_Number_Type loop
                Inventory_IO.Get (Inventory_File,
                                  Inventory(Location,
                                            Fruit,
                                            Tree_Number));
            end loop;
        end loop;
    end loop;

    Close (Inventory_File);

end Elod_Co;
```

```
with Elod_Co; use Elod_Co;
with Text_IO; use Text_IO;
procedure Elod_Co_Report is

    package Results_IO is new Integer_IO (Integer);
    use Results_IO;

    Total_Bushels_in_Hawaii : Integer;
    Total_Bushels_in_Mexico : Integer;
    Total_Bushels_in_VI     : Integer;

    Total_Bushels_of_Company : Integer;


    function Total_Grapefruits is
            new Sum_in_Bushels
                (Location_Type,
                 Fruit_Type range Grapefruits .. Grapefruits);

    function Total_Pineapples is
            new Sum_in_Bushels
                (Location_Type,
                 Fruit_Type range Pineapples .. Pineapples);

    function Total_Limes is
            new Sum_in_Bushels
                (Location_Type,
                 Fruit_Type range Limes .. Limes);

    function Total_in_Hawaii is
            new Sum_in_Bushels
                (Location_Type range Hawaii .. Hawaii,
                 Fruit_Type);

    function Total_in_Mexico is
            new Sum_in_Bushels
                (Location_Type range Mexico .. Mexico,
                 Fruit_Type);

    function Total_in_Virgin_Islands is
            new Sum_in_Bushels
                (Location_Type range
                         Virgin_Islands .. Virgin_Islands,
                 Fruit_Type);

    function Total_Grapefruits_in_Mexico is
            new Sum_in_Bushels
                (Location_Type range Mexico .. Mexico,
                 Fruit_Type range Grapefruits .. Grapefruits);
```

```
    function Trees_Yielding_Less_Than is
            new Sum_of_Trees
                   (Location_Type,
                    Fruit_Type,
                    "<");

    function Lime_Trees_Yielding_More_Than is
            new Sum_of_Trees
                   (Location_Type,
                    Fruit_Type range Limes .. Limes,
                    ">");

    function Trees_in_Virgin_Islands_Yielding is
            new Sum_of_Trees
                   (Location_Type range
                           Virgin_Islands .. Virgin_Islands,
                    Fruit_Type,
                    "=");

begin    -- Elod_Co_Report.

    -- The total yield in bushels of each location and
    -- of the entire company.

    Total_Bushels_in_Hawaii := Total_in_Hawaii;
    Total_Bushels_in_Mexico := Total_in_Mexico;
    Total_Bushels_in_VI := Total_in_Virgin_Islands;

    Total_Bushels_of_Company := Total_Bushels_in_Hawaii +
                                Total_Bushels_in_Mexico +
                                Total_Bushels_in_VI;
```

```
New_Page;
Set_Line (5);
Set_Col (10);
Put ("ELOD FRUIT COMPANY ANNUAL YIELD (in bushels)");
New_Line (3);
Put ("Total bushels grown: ");
Put (Total_Bushels_of_Company);
New_Line;
Put ("Total bushels grown in Hawaii: ");
Put (Total_in_Hawaii);
New_Line;
Put ("Total bushels grown in Mexico: ");
Put (Total_in_Mexico);
New_Line;
Put ("Total bushels grown in the Virgin Islands: ");
Put (Total_in_Virgin_Islands);
New_Line;

-- The total yield in bushels of each fruit.

Put ("Total yield in bushels of grapefruit: ");
Put (Total_Grapefruits);
New_Line;
Put ("Total yield in bushels of pineapples: ");
Put (Total_Pineapples);
New_Line;
Put ("Total yield in bushels of limes: ");
Put (Total_Limes);
New_Line;

-- The total yield in bushels of grapefruit trees in Mexico.

Put ("Total bushels of grapefruit grown in Mexico: ");
Put (Total_Grapefruits_in_Mexico);
New_Line;

-- The number of trees in the company that yield fewer than
-- five bushels of fruit a year.

Put ("Number of trees in the company that yielded fewer");
Put ("than five bushels of fruit: ");
Put (Trees_Yielding_Less_Than (5));
New_Line;

-- The number of lime trees in the company that yield more
-- than fifteen bushels a year.

Put ("Number of lime trees in the company that yielded");
Put ("more than fifteen bushels: ");
Put (Lime_Trees_Yielding_More_Than (15));
New_Line;
```

```
    -- The number of trees in the Virgin Islands that yield
    -- twenty bushels of fruit a year.

    Put ("Number of trees in the Virgin Islands that yielded");
    Put ("twenty bushels of fruit: ");
    Put (Trees_in_Virgin_Islands_Yielding (20));

    New_Page;

end Elod_Co_Report;
```

CHAPTER 4

APPLICATIONS OF DATA ABSTRACTION

EXERCISE 4.1

LINKED LIST APPLICATION: QUEUES

Objective

This exercise introduces three implementations of queues, singly linked, doubly linked, and circularly linked, and demonstrates their use in Ada.

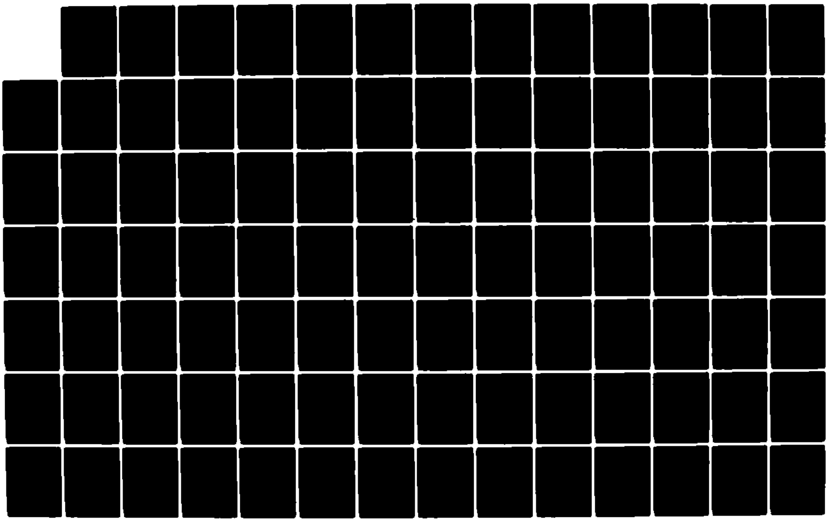
Tutorial

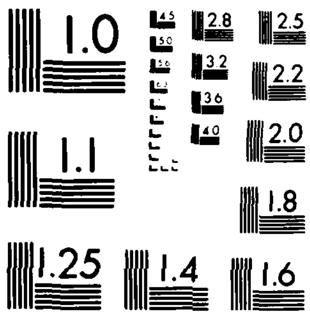
The previous chapter of this book introduced the concept of data abstraction, notably that it entails a consideration of the data structure and also of the operations defined for it, or in other words, of function as well as form. This chapter examines the forms of some fundamental data structures -- queues, and in the next exercise, trees -- and illustrates how they reflect the principles of data abstraction.

A linear list is a set of nodes whose relational scheme is essentially one-dimensional. Typically, the functions that one applies to a list are to gain access to the nth node, to search the list for a particular value, to insert a new node, or to delete a node. There are three common applications of simple linear lists: stacks, deques, and queues. Stacks allow insertions and deletions (generally referred to as "popping" and "pushing") at only one end of the list. The reader is probably familiar with their use. Deques (or double-ended queues) are considerably more general, allowing all insertions and deletions at either end of the list. And queues, the subject of this exercise, have insertions at one end of the list and deletions at the other. (See the figures that follow.)

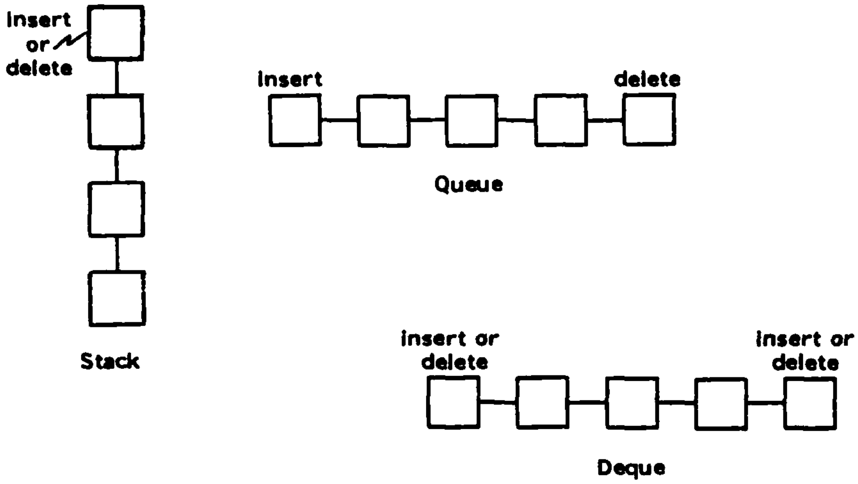
AD-A146 257 ADVANCED ADA (TRADEMARK)(U) SOFTECH INC WALTHAM MA 3/5
JUL 84 DAAB07-83-C-K514

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



The simplest and perhaps most common structure for a queue is an array. A more flexible queue can be achieved, however, by creating a structure in which each node contains a link to the next one. In a linked list queue, or more simply, linked queue, it is easier and more efficient to delete a node -- one merely changes the link. In an array, a deletion may involve moving every succeeding element forward in the queue.

In general, linked lists are used for lists to which there will be frequent changes, or for ordered lists to which there may be some, if only a few, changes. If the elements of the list are stored in some order, insertions, which are made at a position dictated by the order, are far more easily made using a linked list than an array. The linked structure also facilitates the joining of two lists, or the division of a list into two or several separate lists. Finally, the amount of storage occupied by the list is not fixed <u>a priori</u>.

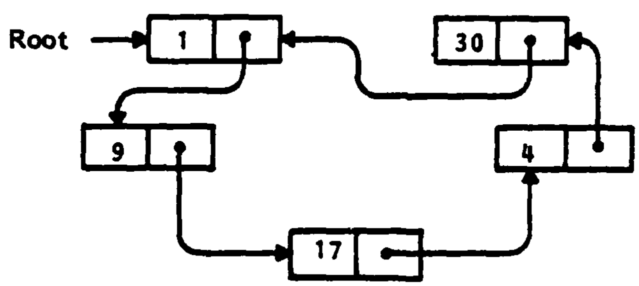
There are three implementations of linked queues: singly linked, doubly linked, and circularly linked. Selecting the type of linked list to use depends upon the circumstances of the problem.

A singly linked queue was used to solve the problem in Exercise 2.3, where the supply order information of a manufacturing company was processed in a priority queue. The structure of the queue was built using a recursive type. Each list cell, or node, was an allocated record having a component that contained an access to the next record:

```
type Node_Type;    -- Incomplete type declaration.
type Pointer_Type is
        access Node_Type;
type Node_Type is
       record
           Data    : Data_Type;
           Pointer : Pointer_Type;
       end record;
```

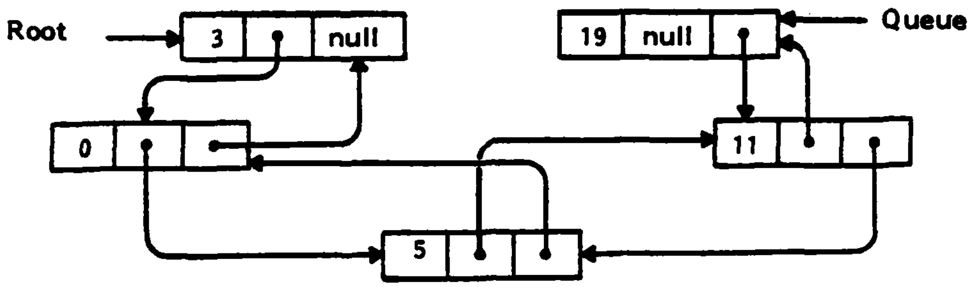
where Data_Type is the type of the data stored in the node.

A circularly linked list has no beginning or end. In terms of the queue drawn above that means that the access component of the last cell would link back to Root. A circular list with a pointer to the first node can be considered similar to a linear list with a pointer to the beginning and end. An illustration of a circular list storing integers follows.



The structure of the circular list enables the list to be divided quite easily. The circular list can be stored in several variables starting from any point in the list because a pointer to any node leads to the entire list. A circular list may be inserted into another circular list with very few statements.

Doubly linked lists are similar to singly linked lists, but in addition to the link to the next node, each node has a link to the previous node. Below is a diagram of a doubly linked list storing integer values. The beginning of the list is marked by Root and the end of the list is marked by Queue.



A doubly linked list may be either linear or circular. Its advantage is that a pointer may travel through the list either forward or backward. The added mobility makes doubly linked lists the most flexible, and the operations applied to the list most simple. The basic operations used for manipulating a doubly linked list are covered in detail in the problem at the end of this exercise.

Consider the use of a singly linked list in implementing the inventory of an engine parts supply store. Each node of the list contains the identification number of the part, the number of that part currently in stock, the first ten letters of its name, and a point… to the next part. The definition of a node for this queue is,

```
type Code_Type is range 100 .. 10_000;
type Number_in_Stock_Type is range 0 .. 200;
```

```
type Node_Type;          -- Incomplete type definition.

type Pointer_Type is
     access Node_Type;

type Node_Type is
   record
      Code              : Code_Type;
      Number_in_Stock   : Number_in_Stock_Type;
      Name_of_Part      : String (1 .. 10);
      Next_Node         : Pointer_Type;
   end record;
```

When a new part is added to the inventory a procedure that adds a node to the queue is required. If one assumes that there is no ordering in this list, nodes are added to the end. The front of the queue is marked by a pointer Root. The end of the queue is marked by a pointer Queue.

```
Root, Queue: Pointer_Type;
```

To "add" a node to the empty list, one writes,

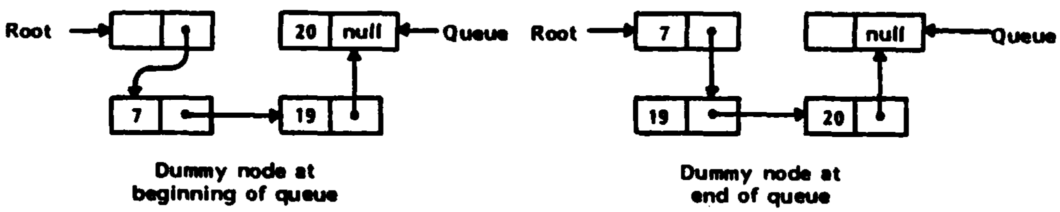
```
Queue := new Node_Type'          -- Node allocated.
         (Code => New_Code,
          Number_in_Stock =>
             New_Number_in_Stock,
          Name => New_Name,
          Next => null);

Root := Queue ;                   -- Queue update.
```

where New_Code, New_Number_in_Stock, New_Name are the values that are to be given to the new node.

Note that the Next component cannot be filled because the next node does not yet exist. When another node is added, the new node is allocated, the Next component of the current node is assigned a pointer to the new node, and then the data components of the new node are filled. Thus, a different sequence of statements must be used for the

first node of a queue than for all of the other nodes. In order to avoid non-uniformity, a dummy node is often placed at the beginning or the end of the queue.

A dummy node is of the same type as other nodes, but it holds no value. Two separate queues, each storing integers, are illustrated in the diagram below, one with a dummy node at the beginning, and the other with it at the end. (Root points to the first node, and Queue points to the last.)



The reason for including a dummy node in a queue structure is essentially that it enables the functions applied to the queue to be written without special boundary conditions for the first or last node. When used, dummy nodes occur at the end of the queue where nodes are added.

A dummy node may be included in a circular linked list and in a doubly linked list. Used in a circular list, it also prevents the list from disappearing when it becomes empty. Alternatively, one can determine the "end" of the list when searching its length by checking for the node at which the search commenced.

To return to the engine parts inventory, a dummy node inserted at the end of the queue could be allocated when Root and Queue are declared. So we write instead:

```
Root  : Pointer_Type := new Node_Type;
Queue : Pointer_Type := Root;
```

Before there is anything in the list, Root and Queue point to a single empty node.

Every node (the first node with data as well as all of the other nodes), is added by (1) filling the dummy node, (2) allocating a new node, (3) assigning a pointer to the new dummy node to the Next component of the current node, and (4) adjusting Queue so that it points to the new dummy node. The code follows:

```
Queue.Code := Code;              -- Fill dummy node.
Queue.Number_in_Stock :=
   Number_in_Stock;
Queue.Name := Name;
Queue.Next := new Node_Type; -- Allocate a new node, put
                             -- pointer to it in Next.
Queue := Queue.Next;         -- Queue points to new dummy.
```

again where Code, Number_in_Stock, and Name are the values given to the new node. Because this sequence of statements leaves a dummy node at Queue, that is, prepared for the addition of another node, the process can be encapsulated in a subprogram and invoked for all additions to the queue, including that of the first node. While it does not appear unduly inefficient to use different code for adding the first node than for adding subsequent nodes, as was shown for the queue without a dummy node, it is really substantially better to use the same process for every addition because the queue could become empty again. The special circumstance of the addition of the first node is really one of the addition of any node to an empty list, which is a situation that may arise often in the lifetime of a queue.

One should take into account the boundary conditions, such as an empty queue, when designing the structure of a linked queue. The inclusion of a dummy node in the engine parts queue enables the programmer to write a subprogram for the addition of nodes, that can, without any consideration of the size of the queue, add nodes ad nauseum. But this is not to say that it is always best to include a dummy node. Their value depends upon the purpose of the queue (read set of functions operating on it).

Searching the queue for an occurrence of an engine part with code number 1226 is accomplished by declaring a temporary pointer,

```
Pointer : Pointer_Type;
```

that moves through the queue, checking each Code component. The code for this search is:

```
Pointer := Root;
while not (Pointer = Queue or
           Pointer.Code = 1226) loop
    Pointer := Pointer.Next;
end loop;
```

Pointer steps through the queue until it reaches either Queue (indicating an unsuccessful search) or a node with the specified code. One could add code to indicate whether a node is found or not:

```
if Pointer /= Queue then ...            -- node found
```

Deletion of a node is accomplished simply by changing the Next component of the previous node. For example, if Pointer now points to the node to be deleted, one (1) obtains a pointer to the previous node, and (2) puts a pointer to the succeeding node in the Next component of that previous node. In order to obtain the previous node, another pointer is declared and the list must again be searched. If the node to be deleted is the first node in the queue (pointed to by Root) then Root is simply advanced to Root.Next. The code follows:

```
declare

    Previous_Node : Pointer_Type := Root;

begin

    if Pointer /= Root then
        while Previous_Node.Next /= Pointer loop
            Previous_Node := Previous_Node.Next;
        end loop;
    else
        Root := Root.Next;
    end if;
    Previous_Node.Next := Pointer.Next;

end;
```

At the conclusion of the deletion, Pointer still points to the deleted node.

Consider again the engine parts. If the queue were ordered, that is, if nodes occur in an order prescribed by, for example, the code of

the part, it could be necessary to add nodes to positions in the list other than Queue. A new node would be inserted by (1) finding the place in the queue where it belongs, (2) allocating a new node, (3) changing the Next component of the preceding node to point to the new node, (4) filling the new node with data, and (5) putting a pointer to the succeeding node in the Next component of the new node. Again, where New_Code, New_Number_in_Stock, and New_Name are objects having the value of the code, the number in stock, and the name of the part, respectively, the insertion is written:

```
while Pointer.Next.Code > Code  -- Find location of insert.
loop
    Pointer := Pointer.Next;
end loop;
Next_Node := Pointer.Next;      -- Save pointer to next node.
Pointer.Next := new Node_Type'  -- Allocate new node
        (Code => New_Code,      -- and fill it.
         Number_in_Stock =>
             New_Number_in_Stock,
         Name => New_Name,
         Next => Next_Node);
```

In the code above, the method for finding the place in the queue where the new node is inserted is a simple step through the list. If an ordered queue is expected to be long, however, a more efficient searching method can be used. Searching algorithms are discussed in Exercise 5.1.

Note that a node of a queue may have any number of any type of data components. The three components used in the engine parts inventory queue could be replaced, for example, by a single access to another record storing all of the data. We could redefine Node_Type to be,

```
type Data_Type is
    record
        Code            : Code_Type;
        Number_in_Stock : Number_in_Stock_Type;
        Name            : String (1 .. 10);
    end record;
type Data_Access is access Data_Type;

type Node_Type is
    record
        Data : Data_Access;
        Next : Pointer_Type;
    end record;
```

The addition to the end of the queue of a node for a distributor, #992, five in stock, would then be,

```
Queue.Data := new Data_Type'(Code => 992,
                             Number_in_Stock => 5,
                             Name => "Distributo");
Queue.Next := new Node_Type;
Queue  := Queue.Next;
```

Using an allocated data part is not necessarily better than storing the data and pointers in the same record. It may be better if there is a lot of data because the data can be moved as a single record aggregate, rather than by individual component. Also, note that an allocated data part enables the implementation of a queue whose elements are of different sizes, a general advantage to linked allocation.

Problem

A flight training center has decided to install an automated system to manage its loans of equipment to student pilots. This system is supposed to support the following functions at a minimum:

- loan an available plane or simulator;
- if the requested equipment is not available, to take the request and fulfill it on a first come, first served basis; and
- when loaned equipment is returned, to place it back on the available list, fulfilling any outstanding requests;

In designing this system, design it so that the program can be easily enhanced. An example of a desirable enhancement (not to be coded in this exercise) would be to add some sort of provision for maintenance. In other words, when equipment is returned, the user may specify that the plane needs an oil change, new tires, whatever. The front desk manager should then be able to send this request to maintenance personnel. Of course, the front desk manager's list of available equipment would reflect which items are currently unavailable due to repairs. Another enhancement (also not required for the purposes of this exercise) would allow the manager to record who requested what equipment and who borrowed what equipment.

Assume that the following equipment is available (this list reflects what is currently on hand - plans have been made to acquire more aircraft and simulators, so your solution should accommodate these anticipated purchases):

- 5 Cessna 150's
- 8 Cessna 172's
- 1 C-130
- 4 F-14 simulators
- 6 F-15 simulators
- 10 sets of introductory ground school videotapes
- 11 sets of advanced navigation videotapes

Solution and Discussion

The nature of the problem suggests that a solution that uses queues is in order. Essentially, the person at the desk who runs this system needs some kind of a queue to monitor both the current borrower of equipment and incoming requests for equipment currently on loan. The fact that the training center intends to acquire more equipment means that an easily expandable data structure such as a queue (in linked list implementation) is appropriate. Thus the basic solution to this problem already needs two queues, one to monitor equipment on hand, and the other to monitor equipment requests. Looking at the enhancements, notice how another queue to handle the maintenance requests would be useful. Given that several queues will be used in building this system, it would be nice to develop a single queue and reuse it for each application as opposed to coding three nearly identical queues. Ada does provide just the facility through its generic units. To summarize, the structure of this solution will be dominated by multiple instantiations of a generic queue package.

The manager program itself will "manage" the equipment by using the package-provided queue operations to take equipment off the available list and to queue requests for borrowed equipment. Let us first develop the queue package to see what these operations will be.

First, let us consider what the generic parameters of this queue package should be. One parameter is required, namely the type of item to be queued. At first thought, it might seem that a generic package might be superfluous here; however, after the enhancements are made, not all the queues will handle exactly the same data. For example, the available list will hold information on planes such as the type of plane or simulator and its serial number. The items on the request and loan queues would consist of the type of equipment desired and the name of the student requesting it. Should the programmer be further directed to modify this program and add the maintenance queue, the

queued information would be the equipment, its serial number, and the reason for maintenance. For simplicity, though, we will allow the same type of information to be put on each queue. Thus we have:

```
generic

    type Item_Type is private;    -- allows us to handle record
                                  --  types easily

package Queue_Package is

    ...

end Queue_Package;
```

The next consideration to be addressed is the kind of linked list to be used in implementing the queue. There are three choices: singly linked, doubly linked and circular. We have opted for the doubly linked list for two reasons. First it will make deletions much easier, and second, the code for doubly linked lists has not yet been presented.

Thought must also be given to what kind of operations are to be provided by this package. A user needs the capability to add an item to the queue and to remove a specified item from the queue:

```
procedure Enqueue (Item : in Item_Type);

procedure Dequeue (Item : in Item_Type);
```

Naturally some provision should be made to notify the user of any attempt to remove an item either from an empty queue or an item which is not currently on the queue. These conditions are best modeled by an exception for each situation:

```
Queue_Empty, Item_Not_Found : exception;
```

In designing a queue package, it is also good programming practice to provide a Boolean function which returns whether or not the queue is empty:

```
function Is_Empty return Boolean;
```

If an array implementation of a queue is being considered, a corresponding function testing for the queue full condition should be written. (Such a function could be provided for the linked list implementation as well - the implementation would return True if an attempt to allocate a new node raised the predefined exception Storage_Error, False otherwise.)

At this point it is worth discussing two design alternatives for the queue data structure to be encapsulated in this package. One design choice includes the queue data type in the package specification and the other hides it entirely in the package body. In the first case, the package exports a queue type which queues objects of the generic formal parameter type. Thus instantiators of the package may create multiple queues which all queue the same type of object. The corresponding package specification would then look like:

```
generic

    type Item_Type is private;

package Queue_Package is

    type Queue_Type is limited private;

    procedure Enqueue (Queue : in Queue_Type;
                       Item  : in Item_Type);
    procedure Dequeue (Queue : in Queue_Type;
                       Item  : in Item_Type);
    function Is_Empty (Queue : in Queue_Type) return Boolean;

    Queue_Empty, Item_Not_Found : exception;

private

    type Node_Type is
       record
           Data : Item_Type;
           Prev : Queue_Type;
           Next : Queue_Type;
       end record;
    type Queue_Type is access Node_Type;

end Queue_Package;
```

Queue_Type is declared to be limited private in order to restrict the user to use only those subprograms provided by the package. An incomplete type declaration for Node_Type in the private part of the package is not needed because Queue_Type, the type of the components Prev and Next, is "known" as the result of its declaration as a limited private type.

A more probable scenario is one in which the user needs only one queue for any particular kind of object. This design, discussed in depth, makes the generic package itself the abstract queue data type. The instantiation of the package creates an empty queue whose contents may be modified through judicious use of the visible subprograms.

The private type declaration can now be deleted from the package specification. The bodies of these subprograms and the implementation of the queue itself should be placed in the package body to conceal them from the user. So the package specification is,

```
generic

    type Item_Type is private;

package Queue_Package is

    procedure Enqueue (Item : in Item_Type);
    procedure Dequeue (Item : in Item_Type);
    function Is_Empty return Boolean;

    Queue_Empty, Item_Not_Found : exception;

end Queue_Package;
```

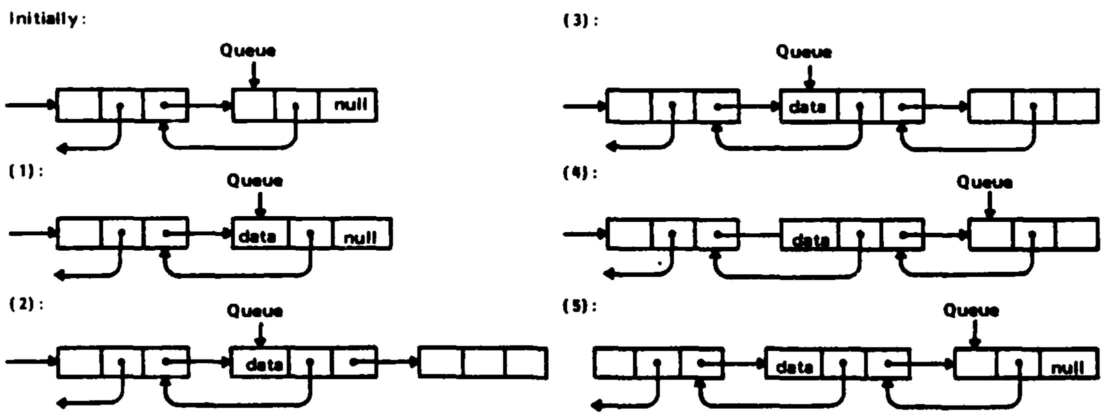
Each node of the queue is of a record type having three components. The first component is the data of type Item_Type. The second component is an access to the previous node, and the third component is an access to the next node. The recursive structure looks like this:

```
type Node_Type;                  -- incomplete type declaration
type Queue_Type is access Node_Type;
type Node_Type is
    record
        Data : Data_Type;
        Prev : Queue_Type;       -- pointer to previous node
        Next : Queue_Type;       -- pointer to next node
    end record;
```

Two pointers of Queue_Type can then be declared to mark the front and end of the queue. It is better to have a dummy node at the end of the queue at which insertions occur (as explained in the tutorial), so the dummy node is allocated at the declaration of Root and Queue. Both Root and Queue are then initialized to point to that node:

```
Root  : Queue_Type := new Node_Type;
Queue : Queue_Type := Root;
```

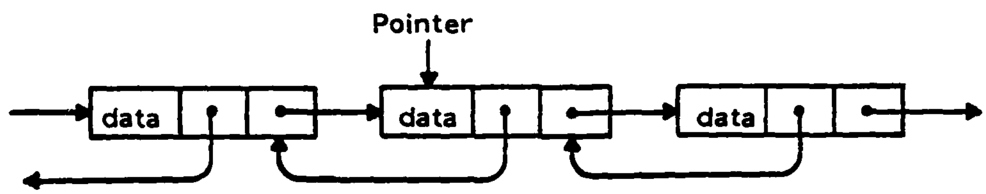
The subprogram Enqueue adds a node with the given data to the end of the queue. It must also allocate a new dummy node and adjust all of the appropriate pointers. The steps involved in the creation of the new node are (1) to put the data in the data component of the current dummy node, (2) to allocate a new node and put a pointer to the new node in the Next component of the current node, (3) to put a pointer to the current node in the Prev component of the newly allocated dummy node, (4) to change Queue to point to the new dummy node, and (5) to put null in the Next component of the new dummy node. An illustration of these five steps follows:



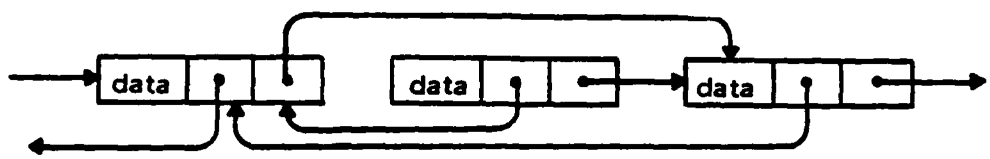
The code for the procedure is:

```
procedure Enqueue (Item : in Item_Type) is
begin

    Queue.Data := Item;               -- Data into dummy node
    Queue.Next := new Node_Type;      -- Pointer to new dummy in Next
    Queue.Next.Prev := Queue;         -- Pointer in prev of new dummy
    Queue := Queue.Next;              -- Queue point to new dummy
    Queue.Next := null;               -- Next of new dummy grounded

end Enqueue;
```

The second subprogram is the procedure Dequeue. It deletes the first node in the queue having the given data. An object is declared to serve as a pointer that proceeds through the list until it either reaches the first node whose data component is the same as the in parameter data or reaches Queue without having found the data. In this second case, it raises the exception Item_Not_Found. Should the queue be empty at the start of the deletion algorithm, the exception Queue_Empty is raised. The deletion is shown pictorially first:



BEFORE DELETION



AFTER DELETION

The procedure is:

```
procedure Dequeue (Item : in Item_Type) is

    Pointer : Queue_Type := Root;

begin   -- Dequeue

    if Root = Queue then
        raise Queue_Empty;  -- exit immediately
    end if;

    while Pointer /= Queue loop
        if Pointer.Data = Item then
            Pointer.Prev.Next := Pointer.Next;
            Pointer.Next.Prev := Pointer.Prev;
            return;    -- deletion accomplished; exit procedure
        end if;
    end loop;

    raise Item_Not_Found;

end Dequeue;
```

Note that were the queue only singly linked, either a second pass through the queue would have to be made in order to obtain the previous node or an additional pointer would have to be declared to save the value of the previous node at each iteration. In circumstances where the queue is very long, or where there are many deletions or causes for obtaining the previous node, the double links are well worth the extra record component. In ordered queues, where searches through the queue may go either forward or backward, by virtue of the order, double links are advantageous to the point of necessity.

The function Is_Empty compares the two access values Root and Queue:

```
function Is_Empty return Boolean is
begin
    return Root = Queue;
end Is_Empty;
```

Having developed the generic queueing package, let us look at the kind of items that have to be queued. The available list is a list of equipment types and their serial numbers:

```
type Equipment_Make_Type is
    (Cessna_150, Cessna_172, C_130, F_14_Simulator,
     F_15_Simulator, Ground_School_Tapes,
     Advanced_Navigation_Tapes);
type Equipment_Type is
    record
        Equipment_Make : Equipment_Make_Type;
        Serial_Number  : String (1 .. 10);
    end record;
```

In order to accommodate new acquisitions, the declaration of Equipment_Make_Type might have to be expanded, depending on the type of acquisition. The master list can be declared as a queue holding items of type Equipment_Type. As new equipment is purchased and delivered, the master list is updated easily:

```
Available_Equipment.Enqueue ((New_Equipment, Serial_Number));
```

where Available_Equipment is an instantiation of the generic queue package and New_Equipment is the item just acquired.

In order to modify this solution to track the names of equipment borrowers and requestors, additional data structures would be defined. Also a useful function to add to the generic queue package would be a search function which, given a key, searches for an item on the queue and returns the whole item. Performing the search on a key rather than the whole record is useful because of the following situation. X checks out a Cessna 150. Y requests the same Cessna 150. When X returns it, the manager wants to find out not only whether that Cessna airplane was requested but also by whom. Since the request queue would hold both the plane and the name, but the plane is the only known parameter, a search routine that searches on a key would be necessary.

To complete the solution, we must create the various queues needed by the front desk manager. The following queues are needed: a list of available equipment (created whenever the system is brought on-line),

updated whenever a request is served, a list of equipment borrowed, and a list of equipment to be loaned out as soon as it becomes available. These queues are now instantiated:

```
package Available_Equipment is new
        Queue_Package (Equipment_Type);
package Borrowed_Equipment is new
        Queue_Package (Equipment_Type);
package Equipment_to_be_Loaned is new
        Queue_Package (Equipment_Type);
```

The body of the manager consists of a loop in which equipment is enqueued and dequeued from the various queues. (Assume that the necessary I/O exists). Requests must be either for a loan or a return of equipment. If the equipment is available, it is loaned out. If the item is not found on the available list, then the exception Item_Not_Found will be raised upon an attempt to dequeue it from that queue. An exception handler is provided in the block nested inside the loop so that this request can be put on the queue for equipment to be loaned out. Whenever equipment is returned, the manager program checks to see whether there is a request for that piece of equipment outstanding. This check is performed by assuming that the piece of equipment has been requested and then dequeueing it from the to-be-loaned list. Should the piece not have been requested, the exception Item_Not_Found would again be raised, and it is handled within the block, in this case by specifically taking no action. The other actions are not anticipated to raise exceptions, therefore no further handlers are provided in the block.

```
loop
    begin
        Get (Request);              -- either a return or a loan
        if Request = Loan then
            Get (Equipment_Request); --includes serial number
            Available_Equipment.Dequeue (Equipment_Request);
            Borrowed_Equipment.Enqueue (Equipment_Request);
        else  -- Request = Return
            Get (Equipment_Returned); --includes serial number
            Borrowed_Equipment.Dequeue (Equipment_Returned);
            Available_Equipment.Enqueue (Equipment_Returned);
            Equipment_to_be_Loaned.Dequeue
                                   (Equipment_Returned);
```

```
                -- if nobody has requested the equipment and this
                --  call raises an exception, nothing more is
                --  moved around the queues.
                Available_Equipment.Dequeue (Equipment_Returned);
                Borrowed_Equipment.Enqueue (Equipment_Returned);
            end if;
        exception
            when Available_Equipment.Item_Not_Found =
                Put_Line ("Equipment requested not available." &
                          "Request has been queued.");
                Equipment_to_be_Loaned.Enqueue
                                  (Equipment_Request);
            when Equipment_to_be_Loaned.Item_Not_Found =
                null;   -- don't worry, the item was never requested
                        -- in the first place
        end;
    end loop;
```

In the complete solution code that follows, the object and type declarations necessary to make the code compilable are provided:

```
generic

    type Item_Type is private;

package Queue_Package is

    procedure Enqueue (Item : in Item_Type);
    procedure Dequeue (Item : in Item_Type);
    function Is_Empty return Boolean;

    Queue_Empty, Item_Not_Found : exception;

end Queue_Package;

package body Queue_Package is

    type Node_Type;                 -- incomplete type declaration
    type Queue_Type is access Node_Type;
    type Node_Type is
        record
            Data : Data_Type;
            Prev : Queue_Type;      -- pointer to previous node
            Next : Queue_Type;      -- pointer to next node
        end record;

    Root  : Queue_Type := new Node_Type;
    Queue : Queue_Type := Root;
```

```
    procedure Enqueue (Item : in Item_Type) is
    begin

        Queue.Data := Item;            -- Data into dummy node
        Queue.Next := new Node_Type;   -- Pointer to new dummy in Next
        Queue.Next.Prev := Queue;      -- Pointer in prev of new dummy
        Queue := Queue.Next;           -- Queue point to new dummy
        Queue.Next := null;            -- Next of new dummy grounded

    end Enqueue;

    procedure Dequeue (Item : in Item_Type) is

        Pointer : Queue_Type := Root;

    begin   -- Dequeue

        if Root = Queue then
            raise Queue_Empty;  -- exit immediately
        end if;

        while Pointer /= Queue loop
            if Pointer.Data = Item then
                Pointer.Prev.Next := Pointer.Next;
                Pointer.Next.Prev := Pointer.Prev;
                return;     -- deletion accomplished; exit procedure
            end if;
        end loop;

        raise Item_Not_Found;

    end Dequeue;

    function Is_Empty return Boolean is
    begin
        return Root = Queue;
    end Is_Empty;

end Queue_Package;

with Queue_Package; use Queue_Package;
with Text_IO; use Text_IO;
procedure Manager is

    type Request_Type is (Loan, Return);

    type Equipment_Make_Type is
        (Cessna_150, Cessna_172, C_130, F_14_Simulator,
         F_15_Simulator, Ground_School_Tapes,
         Advanced_Navigation_Tapes);
```

```
    type Equipment_Type is
        record
            Equipment_Make : Equipment_Make_Type;
            Serial_Number  : String (1 .. 10);
        end record;

    Request            : Request_Type;
    Equipment_Request  : Equipment_Type;
    Equipment_Returned : Equipment_Type;

    package Request_IO is new Enumeration_IO (Request_Type);
    use Request_IO;

    package Available_Equipment is new
                Queue_Package (Equipment_Type);
    package Borrowed_Equipment is new
                Queue_Package (Equipment_Type);
    package Equipment_to_be_Loaned is new
                Queue_Package (Equipment_Type);

begin  -- Manager

    -- set up master list
    Available_Equipment.Enqueue ((Cessna_150, "N19982    "));
    Available_Equipment.Enqueue ((Cessna_150, "N24409    "));
    Available_Equipment.Enqueue ((Cessna_150, "N35974    "));
    Available_Equipment.Enqueue ((Cessna_150, "N25561    "));
    Available_Equipment.Enqueue ((Cessna_150, "N276ZU    "));
    Available_Equipment.Enqueue ((Cessna_172, "N61992    "));
    Available_Equipment.Enqueue ((Cessna_172, "N1257E    "));
    Available_Equipment.Enqueue ((Cessna_172, "N12221    "));
    Available_Equipment.Enqueue ((Cessna_172, "N619UT    "));
    Available_Equipment.Enqueue ((Cessna_172, "N45331    "));
    Available_Equipment.Enqueue ((Cessna_172, "N54323    "));
    Available_Equipment.Enqueue ((Cessna_172, "N40223    "));
    Available_Equipment.Enqueue ((Cessna_172, "N541ZT    "));
    Available_Equipment.Enqueue ((C_130, "T333268   "));
    Available_Equipment.Enqueue ((F_14_Simulator, "0000000001"));
    Available_Equipment.Enqueue ((F_14_Simulator, "0000000002"));
    Available_Equipment.Enqueue ((F_14_Simulator, "0000000003"));
    Available_Equipment.Enqueue ((F_14_Simulator, "0000000004"));
    Available_Equipment.Enqueue ((F_15_Simulator, "0000000001"));
    Available_Equipment.Enqueue ((F_15_Simulator, "0000000002"));
    Available_Equipment.Enqueue ((F_15_Simulator, "0000000003"));
    Available_Equipment.Enqueue ((F_15_Simulator, "0000000004"));
    Available_Equipment.Enqueue ((F_15_Simulator, "0000000005"));
    Available_Equipment.Enqueue ((F_15_Simulator, "0000000006"));
```

```
    Available_Equipment.Enqueue ((Ground_School_Tapes, "G1        "));
    Available_Equipment.Enqueue ((Ground_School_Tapes, "G2        "));
    Available_Equipment.Enqueue ((Ground_School_Tapes, "G3        "));
    Available_Equipment.Enqueue ((Ground_School_Tapes, "G4        "));
    Available_Equipment.Enqueue ((Ground_School_Tapes, "G5        "));
    Available_Equipment.Enqueue ((Ground_School_Tapes, "G6        "));
    Available_Equipment.Enqueue ((Ground_School_Tapes, "G7        "));
    Available_Equipment.Enqueue ((Ground_School_Tapes, "G8        "));
    Available_Equipment.Enqueue ((Ground_School_Tapes, "G9        "));
    Available_Equipment.Enqueue ((Ground_School_Tapes, "G10       "));
    Available_Equipment.Enqueue ((Advanced_Navigation_Tapes, "A01 "));
    Available_Equipment.Enqueue ((Advanced_Navigation_Tapes, "A02 "));
    Available_Equipment.Enqueue ((Advanced_Navigation_Tapes, "A03 "));
    Available_Equipment.Enqueue ((Advanced_Navigation_Tapes, "A04 "));
    Available_Equipment.Enqueue ((Advanced_Navigation_Tapes, "A05 "));
    Available_Equipment.Enqueue ((Advanced_Navigation_Tapes, "A06 "));
    Available_Equipment.Enqueue ((Advanced_Navigation_Tapes, "A07 "));
    Available_Equipment.Enqueue ((Advanced_Navigation_Tapes, "A08 "));
    Available_Equipment.Enqueue ((Advanced_Navigation_Tapes, "A09 "));
    Available_Equipment.Enqueue ((Advanced_Navigation_Tapes, "A010 "));
    Available_Equipment.Enqueue ((Advanced_Navigation_Tapes, "A011 "));

    loop
        begin
            Get (Request);              -- either a return or a loan
            if Request = Loan then
                Get (Equipment_Request);   -- includes serial number
                Available_Equipment.Dequeue (Equipment_Request);
                Borrowed_Equipment.Enqueue (Equipment_Request);
            else  -- Request = Return
                Get (Equipment_Returned);  -- includes serial number
                Borrowed_Equipment.Dequeue (Equipment_Returned);
                Available_Equipment.Enqueue (Equipment_Returned);
                Equipment_to_be_Loaned.Dequeue
                                (Equipment_Returned);
                    -- if nobody has requested the equipment and this
                    --  call raises an exception, nothing more is
                    --  moved around the queues.
                Available_Equipment.Dequeue (Equipment_Returned);
                Borrowed_Equipment.Enqueue (Equipment_Returned);
            end if;
        exception
            when Available_Equipment.Item_Not_Found =>
                Put_Line ("Equipment requested not available." &
                          "  Request has been queued");
                Equipment_to_be_Loaned.Enqueue (Equipment_Request);
            when Equipment_to_be_Loaned.Item_Not_Found =>
                null;    -- don't worry, the Item was never requested
                         -- in the first place
        end;
    end loop;

end Manager;
```

EXERCISE 4.2

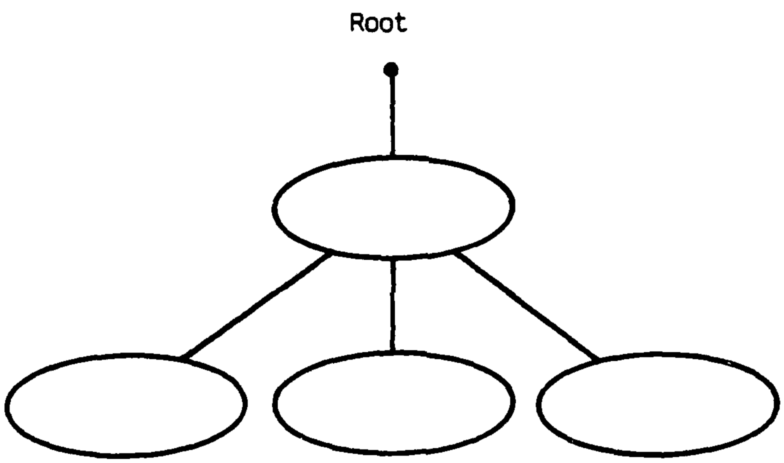
TREE STRUCTURES

Objective

This exercise introduces tree structures and demonstrates their implementation in Ada.

Tutorial

Unlike the linear form of the linked lists discussed in the previous exercise, a tree structure has a branching relationship between the nodes, very much like a natural tree. In the drawing of a tree below,



the upper node is called the root. The nodes descending from a node are called "children" of the node. Conversely, a node's "parent" is the node above it.

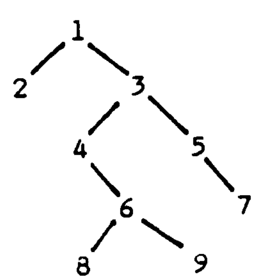
Different kinds of trees are described in terms of the number of children that a node may have. One generally refers to two types of trees, binary and (general) n-ary trees. A binary tree is recursively defined as a finite set of nodes which is either empty or consists of two binary trees called the left and right subtrees. In Ada, the type declaration of a node of a binary tree has a form such as the following:

```
type Node_Type;
type Pointer_Type is
      access Node_Type;
type Node_Type is
   record
      Data  : Data_Type;
      Left  : Pointer_Type;    -- Left child.
      Right : Pointer_Type;    -- Right child.
   end record;
```

where Data_Type is the type of the data stored in the tree. Note that because the definition of a tree is recursive, each node of a tree, other than the root, forms a tree in itself, called a subtree. The number of subtrees that a tree has is called the degree of the tree. Nodes of degree zero are called terminal nodes or leaves, and a non-terminal node is a branch node.

Traversal of a tree is commonly required of functions that manipulate or use tree structures. There are many ways to traverse a binary tree. We discuss three principal traversals here: preorder, postorder, and inorder. In preorder traversal the root is visited first, then the left subtree, and then the right subtree. In postorder traversal, the left subtree is visited first, then the right subtree, and then the root. In inorder traversal (also called symmetric) the left subtree is visited first, then the root, and then the right subtree. Other forms of traversal include breadth-first and bottom-up. Unlike preorder, postorder, and inorder, breadth-first is not inherently recursive. In a breadth-first traversal the top level of the tree is visited, then the next level, etc., until the bottom of the tree is reached. Similarly, a "bottom-up" traversal may traverse one level at a time from the bottom up. A bottom-up traversal may also traverse exactly as a preorder traversal.

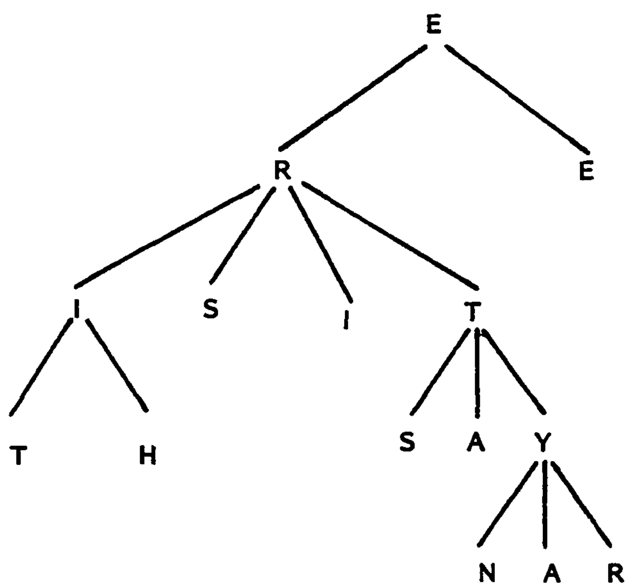
If these methods are applied to the following numeric binary tree,



the following orders of visit are achieved:

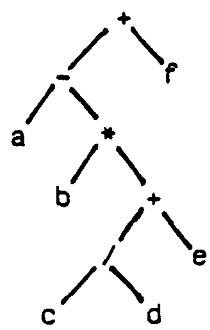
Preorder Traversal:	1 2 3 4 6 8 9 5 7
Postorder Traversal:	2 8 9 6 4 7 5 3 1
Inorder Traversal:	2 1 4 8 6 9 3 5 7
Breadth-First Traversal:	1 2 3 4 5 6 7 8 9

Of the traversal methods discussed above, only the preorder and postorder traversals apply to an n-ary tree. The algorithm is very similar. For a preorder traversal, the root is visited first, then each child subtree, from left to right. In postorder traversal, the child subtrees are visited first, from left to right, and then finally the root. As an example, consider the following tree:



The postorder traversal yields THISISANNARYTREE while the preorder traversal yields ERITHSITSAYNARE.

Consider the evaluation of an algebraic formula. Binary trees are well suited for representing algebraic expressions because each of the operations used, '+', '-', '*', and '/' takes two arguments. In the tree representation the non-terminal nodes of the tree are the operations in the expression, and the left and right nodes are the values to which that operation is applied. For example, the expression a - b * (c / d + e) + f is represented by the following tree,



Note that each subtree forms a "subexpression." The evaluation proceeds roughly from the bottom up. The '/' is applied to c and d, then the '+' is applied to that answer and e, then the '*' is applied to b and that answer, then the '-' is applied to the a and that answer, and then the '+' is applied to that answer and f. Preorder, postorder, and inorder traversals yield linear representations of the expression referred to as prefix, postfix, and infix, respectively. Prefix and postfix are known from their usage in certain pocket calculators. Infix is incorrect without parentheses. The evaluation by each of these methods would appear:

```
Prefix  :  + - a * b + / c d e f
Postfix :  a b c d / e + * - f +
Infix   :  a - b * (c / d + e) + f
```

The type declaration of a node of this tree is,

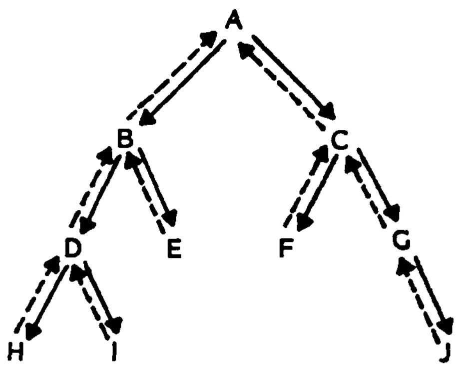
```
type Operator_Type is ('+', '-', '*', '/');

type Node_Type (Is_an_Operator : Boolean);
type Pointer_Type is access Node_Type;
type Node_Type (Is_an_Operator : Boolean) is
    record
        case Is_an_Operator is
            when True =>
                Operator     : Operator_Type;
                Left, Right  : Pointer_Type;
            when False =>
                Expression   : Float;
        end case;
    end record;
```

Notice the use of the discriminant in the record type declaration of a tree node. Closer examination of the tree drawn above shows that the nodes belong to two classes: non-terminal nodes contain an operator and a left and right child, while leaves contain a numeric value. The discriminant allows the node to take the form of the appropriate variant, based on whether or not the node in question is an operator.

The evaluation is performed by a function which recursively applies the operation at a node to the left and right subtrees. The function, called Evaluation here, invokes itself for the evaluation of the left and right subtrees. It is written,

```
function Evaluation (Node : Pointer_Type)
                     return Float is
begin
    if Node.Is_an_Operator then          -- Operator at this node?
        case Node.Operator is            -- Yes, perform operation.
            when '+' =>
               return Evaluation (Node.Left) + Evaluation (Node.Right);
            when '-' =>
               return Evaluation (Node.Left) - Evaluation (Node.Right);
            when '*' =>
               return Evaluation (Node.Left) * Evaluation (Node.Right);
            when '/' =>
               return Evaluation (Node.Left) / Evaluation (Node.Right);
        end case;
    else
        return Node.Expression;          -- No, return value in node.
    end if;
end Evaluation;
```

Alternatively, one can "thread" a tree by adding extra links which point to the next node to be visited. For example, in the tree below



the solid lines represent original links, and the dotted lines represent threads for an inorder traversal. Here the threads point directly to the node's predecessor according to inorder traversal. If a tree is to be traversed many times and in the same directions, one can traverse the tree once to add threads, and then simply use those threads to backtrack during the rest of the program.

Trees are one of the most widely used data structures. They are employed in compilers to parse expressions and in games, where each node represents a move, and a node's children represent the possible successive moves. They are also used to represent some ordering in a list, such as a dictionary, and in that capacity, greatly reduce the time required to locate any given entry, as will be seen in the following chapter on searching and sorting algorithms.

Problem

Given the following tree structure develop a program to traverse a tree in preorder fashion and one to traverse a tree in postorder fashion. As each node is visited, the program should print the data stored in it.

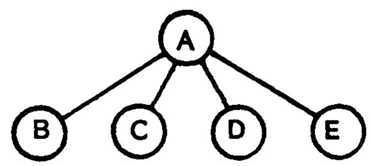
```
package Tree_Specifications is

    -- Incomplete type declaration with discriminant.
    type Multi_Node_Tree (Descendants : Positive := 1);

    -- The pointer to Nodes type.
    type Tree_Type is access Multi_Node_Tree;

    -- An array of these pointers to be used in the record.
    type Descendant_Type is array (Positive range <>) of Tree_Type;

    -- The type of each node.
    type Multi_Node_Tree (Descendants : Positive := 1) is
        record
            Data        : Character;
            Descendants : Descendant_Type (1 .. Descendants);
        end record;

end Tree_Specifications;
```

Discussion and Solution

First, note that because the Descendants component in the tree structure is an array, each node of the tree will have several descendants. And because the length of the Descendants component depends on the value of the discriminant of the node, each node could have a different number of descendants.

Also, note that each node will always have at least one Descendants component because the discriminant that determines the length of the Descendants array is of type Positive. So the smallest possible range of the Descendants array is range 1..1. In nodes that are leaves of the tree this one Descendants component will be null.

Recall that in a postorder tree traversal, a node is visited after all of its descendants have been visited. So in the algorithm for the postorder traversal, the first step is to traverse the descendants. Using the following tree,



a postorder traversal will visit the nodes in the following order:

B C D E A

There are two possible approaches for this traversal of the descendants. One is to trace through the tree keeping track of the current subtree and the parent of the subtree so that once all the descendants are visited, we can backtrack to the parent and eventually to the parent's parent. The other method is to use recursion to trace through the tree.

The first method would require additional access objects and possibly a modification to the tree type to add a thread back to the parent. We will use recursion for the solution because it lends itself well to the traversal of trees. Further, it requires no additional objects or modifications to the existing type. Thus, the program to traverse a tree in postorder order appears:

```
with Tree_Specifications; use Tree_Specifications;
with Text_IO; use Text_IO;
procedure Postorder_Traversal (Tree : in Tree_Type) is

    -- Recursive procedure which traverses a tree in postorder order.

begin -- Postorder_Traversal.

    if Tree /= null then

        -- Process all descendants.

        for I in Tree.Descendants'Range
        loop

            -- Recursively call Postorder_Traversal.

            Postorder_Traversal (Tree.Descendants(I));

        end loop;

        -- Visit the node.

        Put (Tree.Data);

    end if;

end Postorder_Traversal;
```

Let's convince ourselves that this algorithm works. When the procedure is first called, Tree points to the Node A. The program first verifies that a node to process exists. If so, a loop which processes the descendants is entered. So for I in 1 .. 4, the program loops and the procedure is called again with Tree.Descendant (I).

Now Tree points to Node B. Again we make sure a node exists and if so the descendants are processed. So here, again the loop is entered (this time for I in 1..1), and the procedure is called with Tree.Descendants(I).

Now Tree is null and we return from the procedure call without doing anything. Where do we return to? We return to just after the call to the procedure with node B's descendant, which was in a loop from 1 to 1. So we exit the loop, visit node B (B is printed), and return from this procedure call (i.e., the call where Tree was node A's

descendants(I)). Now I is incremented to 2 and the procedure is called again, this time with Tree.Descendants(I) pointing to node C.

Continuing in this fashion until all the descendants of node A are processed, we then fall out of the loop, visit node A (print A), and return from the procedure, i.e., return from the original call of the procedure.

So for the above tree when Postorder_Traversal is called:

B C D E A

will be output.

Now let's take a look at how a preorder traversal would be implemented. Recall that in a preorder traversal, the node is first visited, and then the descendants of the node are visited. Again we use a recursive algorithm.

```
with Tree_Specifications; use Tree_Specifications;
with Text_IO; use Text_IO;
procedure Preorder_Traversal (Tree : in Tree_Type) is

    -- Recursive procedure which traverses a tree in preorder order.

begin -- Preorder_Traversal.

    if Tree /= null then

        -- Visit the current node.

        Put (Tree.Data);

        -- Process the descendants.

        for I in Tree.Descendants'Range
        loop

            -- Recursively call Preorder_Traversal.

            Preorder_Traversal (Tree.Descendants(I));

        end loop;

    end if;

end Preorder_Traversal;
```

Using the same tree as above, a preorder traversal will visit the nodes of the tree in the following order:

A B C D E

Let us convince ourselves that this algorithm works. The program is entered with Tree pointing to A. After checking that a node exists, the node is visited (A is printed) and the Descendants are traversed. The loop is entered (with I having range 1 .. 4), and the procedure is called with Descendants(I).

Now, Tree points to node B. After checking that the node exists, the node is visited (B is printed) and its descendants are traversed. The loop is entered with index range 1..1 and the procedure is called with B's Descendants(I).

Here, the procedure is entered with a null value, so execution returns to the caller. On returning, the loop is exited and execution returns again to the caller (in this case to the call in A's descendant loop). In this loop, I is incremented and the procedure is called with Descendants(I) pointing to node C.

When all A's descendants have been visited, the loop is exited and the program completes execution. The nodes were printed in the following order:

A B C D E

which verifies that the algorithm traverses the tree in preorder order.

The complete code for the problem follows:

```
with Tree_Specifications; use Tree_Specifications;
with Text_IO; use Text_IO;
procedure Postorder_Traversal (Tree : in Tree_Type) is

    -- Recursive procedure which traverses a tree in postorder order.

begin                        -- Postorder_Traversal.

    if Tree /= null then -- Process all descendants.

        for I in Tree.Descendants'Range loop

            -- Recursively call Postorder_Traversal.

            Postorder_Traversal (Tree.Descendants(I));

        end loop;

        -- Visit the node.

        Put (Tree.Data);

    end if;

end Postorder_Traversal;

with Tree_Specifications; use Tree_Specifications;
with Text_IO; use Text_IO;
procedure Preorder_Traversal (Tree : in Tree_Type) is

    -- Recursive procedure which traverses a tree in preorder order.

begin                            -- Preorder_Traversal.

    if Tree /= null then         -- Visit the current node.

        Put (Tree.Data);

        -- Process the descendants.

        for I in Tree.Descendants'Range loop

            -- Recursively call Preorder_Traversal.

            Preorder_Traversal (Tree.Descendants(I));

        end loop;

    end if;

end Preorder_Traversal;
```

CHAPTER 5

CLASSIC APPLICATIONS

EXERCISE 5.1

SEARCHING ALGORITHMS

Objective

This tutorial demonstrates how each of three principal searching algorithms can be used to search an ordered list.

Tutorial

Consider a telephone directory with names listed in alphabetical order. An array of listings (excluding the addresses) is defined,

```
subtype Name_Type is String range (1 .. 17);
subtype Phone_Number_Type is String range (1 .. 7);
type Listing_Type is
    record
        Name   : Name_Type;
        Number : Phone_Number_Type;
    end record;
type Phone_Directory_Type is
        array (Positive range <>) of Listing_Type;
```

A linear search of the Cambridge directory,

```
Cambridge_Directory : Phone_Directory_Type
                      (1 .. 50_000);
```

for the telephone number of a given person's name,

```
Number_to_be_Found  : Phone_Number_Type;
Name_to_be_Found    : Name_Type := "Arnold Schoenberg";
```

is then performed by stepping through the array sequentially. If the listing is not found, an exception is raised:

```
No_Listing : exception;
```

The function that returns the telephone number of the specified name consists essentially of a simple loop. It is written,

```
function Linear_Search_of_Directory
            (Name_to_be_Found : Name_Type)
            return Phone_Number_Type is

begin  -- Linear_Search_of_Directory.

   for Index in Cambridge_Directory'Range loop

      if Cambridge_Directory(Index).Name =
                              Name_to_be_Found then
         return Cambridge_Directory(Index).Number;
      end if;
   end loop;

   raise No_Listing;

end Linear_Search_of_Directory;
```

The average number of steps that it takes to find a listing by using a linear search is half the length of the list which, in the case of the Cambridge telephone directory, is an unsightly twenty-five thousand. A linear search is the only way to search an unordered list; for ordered lists it should be used only when the list is small and infrequently searched. Large ordered lists can alternatively be searched using algorithms such as binary search in which the order facilitates the search.

A binary search jumps to a mid-point in an ordered list, instead of searching the names of the directory consecutively. If the entry at midpoint is determined not to be the name required, the search jumps again to the halfway point in the alphabetical direction of the name. The search continues to jump half the length of that portion of the directory that has not been determined to be alphabetically before or after the required name. The diagram below shows a sample directory with 9 names. The arrow shows the steps taken by the search to find the name Moss.

Auden Eliot Hopkins Lindsay Milton Moss Plath Pound Stevens

The implementation of a binary search requires three objects to mark the current left and right ends, and the midpoint between those ends in the unsearched region.

Assuming that the declarations above for the Cambridge directory are global, the code for the search is written:

```
function Binary_Search_of_Directory
                        (Name_to_be_Found : Name_Type)
                        return Phone_Number_Type is

    Left_End  : Positive := Cambridge_Directory'First;
    Right_End : Positive := Cambridge_Directory'Last;
    Mid_Point : Positive range Cambridge'Range;

begin  -- Binary_Search_of_Directory;

    while Left_End <= Right_End loop

        Mid_Point := (Right_End - Left_End) / 2;

        if Name_to_be_Found < Cambridge_Directory(Mid_Point).Name then
            Right_End := Mid_Point - 1;     -- Name earlier in list.

        elsif
            Name_to_be_Found > Cambridge_Directory(Mid_Point).Name then
            Left_End := Mid_Point + 1;      -- Name later in list.

        else  -- Name_to_be_Found = Cambridge_Directory(Mid_point).Name
            return Cambridge_Directory(Midpoint).Number;

        end if;
    end loop;
    raise No_Listing;               -- Name not in directory.

end Binary_Search_of_Directory;
```

The search continues until the right and left ends cross or until the name to be found is located. At each iteration of the search either the left end or the right end is adjusted to the immediate left or right of the current midpoint, and the midpoint is again calculated.

The average number of searches for a binary search is log n where n is the length of the list. It is best used for reasonably short lists of a fixed length that are searched frequently.

The third method of search that we will examine uses a tree. Instead of an array, as in the examples above, the directory is stored in a binary tree known in this context as a search tree. The type declarations for the tree structure follow:

```
type Node_Type;
type Directory_Pointer_Type is access Node_Type;
type Node_Type is
    record
        Listing : Listing_Type;
        Left    : Directory_Pointer_Type;
        Right   : Directory_Pointer_Type;
    end record;

Cambridge_Directory : Directory_Pointer_Type;
```

Notice that the length of the Cambridge directory need not be specified in order to declare an object of the directory type. Search trees are best used when the number of entries in a list is large and not fixed.

The tree is constructed by adding nodes in a location that corresponds to the order. The procedures that add listings and find listings start searching the tree from the root, and at each node, follow the left subtree if the listing in question is alphabetically before the current node, and the right subtree if the listing in question is alphabetically after the current node. The procedure that adds listings to the Cambridge Directory is written:

```
procedure Add_Listing_to_Directory
                (Directory_Pointer : in out Directory_Pointer_Type;
                 New_Name          : in Name_Type;
                 New_Number        : in Phone_Number_Type) is

begin

    if Directory_Pointer = null then
        Directory_Pointer :=
                new Directory_Pointer_Type'((Name => New_Name,
                                             Number => New_Number),
                                            Left => null,
                                            Right => null);

    elsif New_Name < Directory_Pointer.Listing.Name then
        Add_Listing_to_Directory
                (Directory_Pointer.Left, New_Name, New_Number);

    elsif New_Name > Directory_Pointer.Listing.Name then
        Add_Listing_to_Directory
                (Directory_Pointer.Right, New_Name, New_Number);

    else Directory_Pointer.Listing.Number := New_Number;

    end if;

end Add_Listing_to_Directory;
```

If the new name to be added to the directory is already there, this procedure updates the telephone number of that person.

To find a listing in the directory tree, one uses a very similar algorithm:

```
procedure Find_Telephone_Number
                    (Directory_Pointer: in Directory_Pointer_Type;
                     Name_to_be_Found : in Name_Type;
                     Phone_Number     : out Phone_Number_Type) is

begin

    if Cambridge_Directory = null then raise No_Listing;

    elsif Name_to_be_Found < Directory_Pointer.Listing.Name then
        Find_Telephone_Number (Directory_Pointer.Left,
                               Name_to_be_Found);

    elsif Name_to_be_Found > Directory_Pointer.Listing.Name then
        Find_Telephone_Number (Directory_Pointer.Right,
                               Name_to_be_Found);

    else -- Name_to_be_Found = Directory_Pointer.Listing.Name
        Phone_Number := Directory_Pointer.Listing.Number;

    end if;

end Find_Telephone_Number;
```

The performance of search trees is generally better than that of a binary search, but it is only substantially better if the list is very long and if the tree being searched is balanced.

Problem

Write a package that, using a simple hashing table, allows the user to add a new listing to the Cambridge Directory and to get a person's phone number and address from the Cambridge Directory. Handle the situation where the name is not listed. Use a simple hashing function that adds the numeric values of each letter in the name.

(A hash table is a table of pointers to linked lists containing the actual data. In order to retrieve data (or to determine where to store data), the hashing function is applied to the pertinent part of the data. The value thus calculated is used as the index into the hash table, and the data is accessed by dereferencing the pointer stored at this location in the table. When different items of data hash to the same index (known as a collision), then these items are stored in a linked list, with the indexed hash table location pointing to the beginning of the list.)

Solution and Discussion

The package specification contains the declarations that must be visible to the user, which are the types of the name, phone number, and address for a person's directory listing, and the procedures that add a new listing and find the number and address corresponding to a given person. The name, telephone number, and address types are all subtypes of predefined String:

```
subtype Name_Type is String range 1 .. 17;
subtype Phone_Number_Type is String range 1 .. 7;
subtype Address_Type is String range 1 .. 25;
```

The specifications of the procedures Add_Listing and Find_Listing then appear:

```
procedure Add_Listing (New_Name    : in Name_Type;
                       New_Number  : in Phone_Number_Type;
                       New_Address : in Address_Type);

procedure Find_Listing (Name    : in Name_Type;
                        Number  : out Phone_Number_Type;
                        Address : out Address_Type);
```

If the name with which Find_Listing is called is not in the directory, an exception should be raised. The exception is declared in the visible part of the package:

```
No_Listing : exception;
```

The hash table can be declared in the package body. It is an array of chain-linked telephone listings, that is, each element of the hash array is a linked list of listings. Each telephone listing is of the type,

```
type Listing_Type;
type Hash_Table_Entry_Type is access Listing_Type;
type Listing_Type is
    record
        Name       : Name_Type;
        Number     : Phone_Number_Type;
        Address    : Address_Type;
        Next_Entry : Hash_Table_Entry_Type;
    end record;
```

The length of the hash table should be a large prime number, such as 3019, to avoid collisions as much as possible. So the hash table definitions appear:

```
Hash_Table_Size : constant Integer := 3019;
type Hash_Table_Type is
      array (1 .. Hash_Table_Size)
            of Hash_Table_Entry_Type;
Hash_Table : Hash_Table_Type
      := (1 .. Hash_Table_Size => null);
```

Both Add_Listing and Find_Listing will require a hashing function that will determine the entry in the hash table for a given name. The function adds the numeric value of each character in the name, then multiplies that number by the radix, which is set at one greater than the highest character value, and then mods by the length of the hash table, 3019. The multiplication of the number by the radix has been discovered to reduce the number of collisions. The hash function appears:

```
function Hash (Name : Name_Type)
          return Positive    is

   Hash_Value      : Positive := 1;
   Character_Value : Positive;
   Radix           : constant Positive := Character'Last + 1;

begin

   for Index in Name_Type'Range
      loop
      Character_Value := Character'Pos(Name(Index));
      Hash_Value :=
            (Hash_Value * Radix + Character_Value)
                  mod Hash_Table_Size;
   end loop;

end Hash;
```

Add_Listing is quite simple. It calls Hash to find the entry in the hash table for the new name. Then the end of the linked listings is

found, and the new listing is appended. The procedure follows:

```
procedure Add_Listing (New_Name    : in Name_Type;
                       New_Number  : in Phone_Number_Type;
                       New_Address : in Address_Type)     is

    Pointer  : Hash_Table_Entry_Type;

    begin -- Add_Listing

        Pointer := Hash_Table(Hash (New_Number));
        while Pointer.Next_Entry /= null loop
            Pointer := Pointer.Next_Entry;
        end loop;
        Pointer.Next_Entry :=
       new Hash_Table_Entry_Type'(Name        => New_Name,
                                  Number      => New_Number,
                                  Address     => New_Address,
                                  Next_Entry  => null);

end Add_Listing;
```

Similarly, Find_Listing calls Hash for the entry in the Hash_Table, and then searches the linked list at that entry for the given name. When it is found the phone number and address are returned. If it is not in the list, an exception is raised.

Find_Listing follows:

```
procedure Find_Listing (Name    : in Name_Type;
                        Number  : out Phone_Number_Type;
                        Address : out Address_Type)     is
    Pointer : Hash_Table_Entry_Type;
begin -- Find_Listing.
    Pointer := Hash_Table(Hash (Name));
    while (Pointer.Next_Entry /= null) or
             (Pointer.Name /= Name)
    loop

        Pointer := Pointer.Next_Entry;
    end loop;
    if Pointer.Name = Name then
        Number := Pointer.Number;
        Address := Pointer.Address;
    else
        raise No_Listing;
    end if;
end Find_Listing;
```

The complete solution follows:

```
package Phone_Directory_Hash is

    subtype Name_Type is String range 1 .. 17;

    subtype Phone_Number_Type is String range 1 .. 7;

    subtype Address_Type is String range 1 .. 25;

    No_Listing : exception;

    procedure Add_Listing (New_Name    : in Name_Type;
                           New_Number  : in Phone_Number_Type;
                           New_Address : in Address_Type);

    procedure Find_Listing (Name    : in Name_Type;
                            Number  : out Phone_Number_Type;
                            Address : out Address_Type);

end Phone_Directory_Hash;


package body Phone_Directory_Hash is

    type Listing_Type;

    type Hash_Table_Entry_Type is access Listing_Type;

    type Listing_Type is
        record
            Name       : Name_Type;
            Number     : Phone_Number_Type;
            Address    : Address_Type;
            Next_Entry : Hash_Table_Entry_Type;
        end record;

    Hash_Table_Size : constant Integer := 3019;

    type Hash_Table_Type is
            array (1 .. Hash_Table_Size) of Hash_Table_Entry_Type;

    Hash_Table : Hash_Table_Type := (1 .. Hash_Table_Size => null)
```

```
function Hash (Name : Name_Type)
              return Positive    is

    Hash_Value      : Positive := 1;
    Character_Value : Positive;
    Radix           : constant Positive := Character'Last + 1;

begin

    for Index in Name_Type'Range
    loop
        Character_Value := Character'Pos(Name(Index));
        Hash_Value :=
                (Hash_Value * Radix + Character_Value)
                        mod Hash_Table_Size;
    end loop;

    return Hash_Value;

end Hash;

procedure Add_Listing (New_Name    : in Name_Type;
                       New_Number  : in Phone_Number_Type;
                       New_Address : in Address_Type)     is

    Pointer  : Hash_Table_Entry_Type;

begin  -- Add_Listing

    Pointer := Hash_Table(Hash (New_Number));

    while Pointer.Next_Entry /= null loop
        Pointer := Pointer.Next_Entry;
    end loop;

    Pointer.Next_Entry :=
   new Hash_Table_Entry_Type'(Name       => New_Name,
                              Number     => New_Number,
                              Address    => New_Address,
                              Next_Entry => null);

end Add_Listing;
```

```
procedure Find_Listing (Name    : in Name_Type;
                        Number  : out Phone_Number_Type;
                        Address : out Address_Type)    is

    Pointer : Hash_Table_Entry_Type;

begin  -- Find_Listing.

    Pointer := Hash_Table(Hash (Name));

    while (Pointer.Next_Entry /= null) or
              (Pointer.Name /= Name)
    loop
        Pointer := Pointer.Next_Entry;
    end loop;

    if Pointer.Name = Name then
          Number := Pointer.Number;
          Address := Pointer.Address;
      else
          raise No_Listing;
      end if;

  end Find_Listing;

end Phone_Directory_Hash;
```

EXERCISE 5.2

SORTING ALGORITHMS

Objective

This exercise demonstrates two sorting algorithms: quicksort and heapsort.

Tutorial

A "quicksort" sorts a list by determining a left partition, a dividing element, and a right partition. The dividing element is greater than or equal to every element in the left partition, and less than or equal to every element in the right partition. The quicksort is then recursively applied to the left and right partitions.

Consider the list of officers in the marine corps. Each name is of the type,

```
subtype Officer_Name_Type is String (1 .. 15);
Officer_Name : Officer_Name_Type;
```

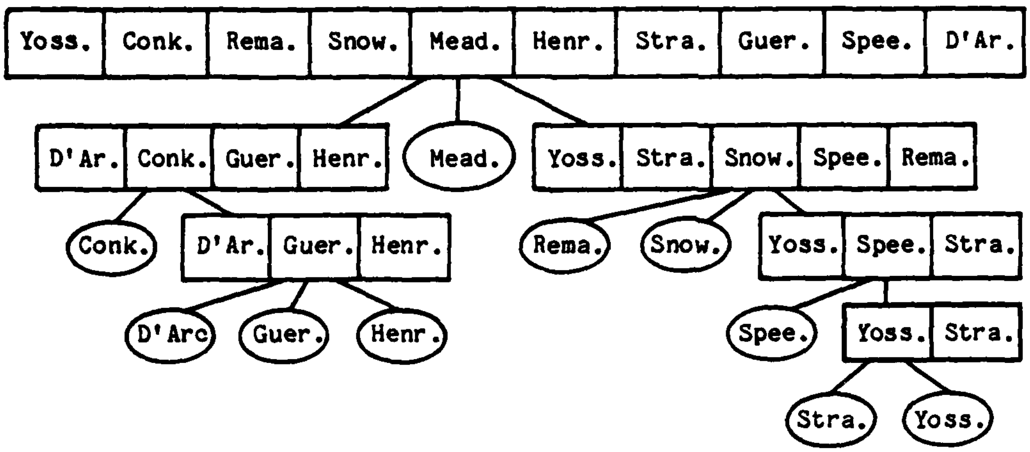
Given an array type for storing the list of names,

```
type Officer_List_Type is
        array (Integer range <>) of Officer_Name_Type;
```

we can declare, for the purpose of demonstrating the sort, a list of ten names,

```
Marine_Officers : Officers_List_Type (1 .. 10) :=
           ("Mike Yossarian ",
            "Jim Conklin    ",
            "Erich Remarque ",
            "William Snowden",
            "George Meade   ",
            "Catherine Henry",
            "Fritz Strassman",
            "Joe Guernica   ",
            "Albert Speer   ",
            "Jeanne D'Arc   ");
```

A quicksort creates a tree from the array, as illustrated below,



The tree, read inorder, yields the sorted list.

The procedure that performs the quicksort for the list of Marine officers recursively divides a list of names in two, such that every element in the left partition is less than or equal to the splitting element, and every element in the right partition is greater than or equal to the splitting element. Marine_Quick_Sort divides and conquers as follows:

```
procedure Marine_Quick_Sort
     (List : in out Officer_List_Type) is

   procedure Split                                      -- Splits list,
       (List           : in out Officer_List_Type;      -- returns split
        Split_Position : out Positive) is separate;     -- position.

   procedure Sort (List : in out Officer_List_Type) is

      Splitting_Position : Positive;

   begin  -- Sort.

      Split (List, Spliting_Position);          -- Split list.

      declare
         Left_Partition  : Officer_List_Type    -- New left part.
               renames List
                     (List'First .. Splitting_Position - 1);
         Right_Partition : Officer_List_Type    -- New right part.
               renames List
                     (Splitting_Position + 1 .. List'Last);

      begin  -- Block statement.

         if Left_Partition'Length > 1 then      -- If left > 1, sort.
            Sort (Left_Partition);
         end if;

         if Right_Partition'Length > 1 then     -- If right > 1, sort
            Sort (Right_Partition);
         end if;

      end;  -- Block statement.

   end Sort;

begin  -- Marine_Quick_Sort.

   if List'Length > 1 then                       -- If list > 1, sort.
      Marine_Quick_Sort (List);
   end if;

end Marine_Quick_Sort;
```

The actual split of a list is performed by the stubbed out procedure Split. Splitting a list involves several iterative steps: (1) assigning the value at the center of the list to the splitting value, (2) filling the space left at the center with the value of the leftmost

element, (3) moving to the space at the leftmost position the value of the first element encountered from the right that is less than splitting value, (4) moving to the space left by that element the first element from the left whose value is greater than the splitting value. These last steps are repeated until the left and right indices either meet or cross one another. Each step of the split of the Marine officers list is illustrated below (top is the "left" from the earlier diagram, the bottom is the "right"):

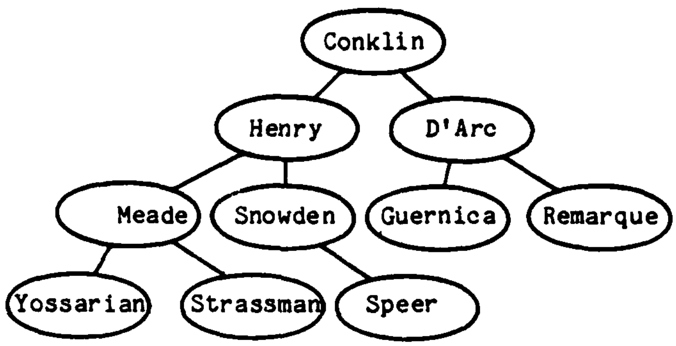
```
(1)  Yossarian     (2)  Yossarian     (3)  <empty>     (4)  D'Arc
     Conklin            Conklin             Conklin         Conklin
     Remarque           Remarque            Remarque        Remarque
     Snowden            Snowden             Snowden         Snowden
     Meade              <empty>             Yossarian       Yossarian
     Henry              Henry               Henry           Henry
     Strassman          Strassman           Strassman       Strassman
     Guernica           Guernica            Guernica        Guernica
     Speer              Speer               Speer           Speer
     D'Arc              D'Arc               D'Arc           <empty>

   Splitting          Splitting
   Value:             Value: Meade

(5)  D'Arc         (6)  D'Arc         (7)  D'Arc       (8)  D'Arc
     Conklin            Conklin            Conklin          Conklin
     <empty>            Guernica           Guernica         Guernica
     Snowden            Snowden            <empty>          Henry
     Yossarian          Yossarian          Yossarian        Yossarian
     Henry              Henry              Henry            <empty>
     Strassman          Strassman          Strassman        Strassman
     Guernica           <empty>            Snowden          Snowden
     Speer              Speer              Speer            Speer
     Remarque           Remarque           Remarque         Remarque

(9)  D'Arc                         (10)  D'Arc
     Conklin                             Conklin
     Guernica                            Guernica
     Henry               Splitting       Henry
     <empty>             Position ->     Meade
     Yossarian                           Yossarian
     Strassman                           Strassman
     Snowden                             Snowden
     Speer                               Speer
     Remarque                            Remarque

   Splitting
   Value: Meade
```

The code for the split follows:

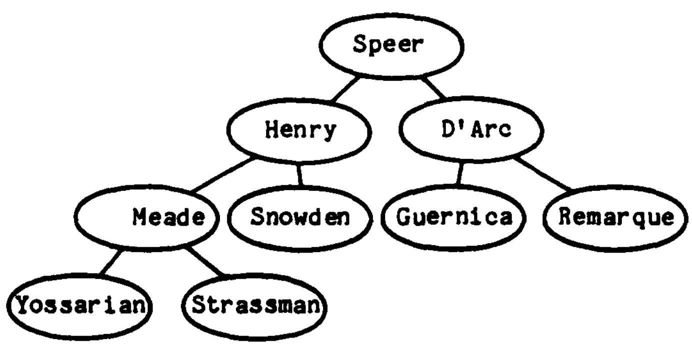
```
separate (Marine_Quick_Sort)
procedure Split (List               : in out Officer_List_Type;
                 Splitting_Position : out Positive) is

    Center          : constant Positive :=
                                (List'First + List'Last) / 2;
    Splitting_Value : constant Officer_Name_Type := List(Center);
    Left            : Positive := List'First;
    Right           : Positive := List'Last;
    Split_Index     : Positive;

begin  -- Split.

    List(Center) := List'First;
    loop
        while Splitting_Value < List(Right)  -- Find rightmost
                and Right > List'First loop  -- value < split_val.
            Right := Right - 1;
        end loop;

        if Left >= Right then                -- Check that Left and
            Split_Index := Left;             -- Right not crossed.
            exit;                            -- If so, exit & finish.
        end if;

        List(Left) := List(Right);           -- Replace Left with
        Left := Left + 1;                    -- Right, incr. Left.

        while List(Left) < Splitting_Value   -- Find leftmost
                and Left < List'Last loop    -- value > split_val.
            Left := Left + 1;
        end loop;

        if Left >= Right then                -- Check that Left and
            Split_Index := Right;            -- Right not crossed.
            exit;                            -- If so, exit & finish.
        end if;

        List(Right) := List(Left);           -- Replace Right with
        Right := Right - 1;                  -- Left, decr. Right.

    end loop;

    List(Split_Index) := Splitting_Value;    -- Return split_index
                                             -- to list.
    Splitting_Position := Split_Index;

end Split;
```

The quicksort is very efficient and (surprise) quick. In its worst possible case, however, a quick sort can take as many as order(n) iterations to sort a list of n elements. A heap sort, while generally a little slower than a quick sort, is not as bad in the worst case.

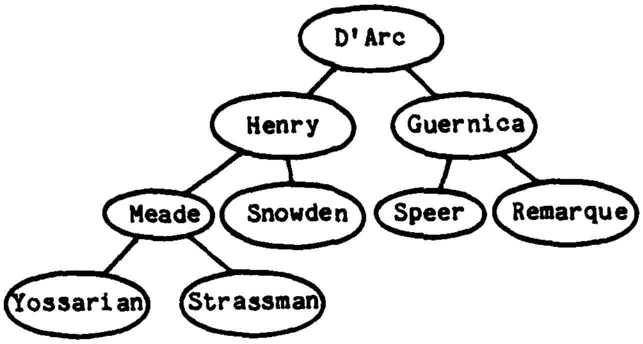
A heap, used for "heapsort" of the Marine officers' names, is a balanced tree in which the name at every node is alphabetically before both of its children. A heapsort will first build a heap of the names by repeatedly adding a name and adjusting its position so that the heap actually remains a heap. When the heap has been built, names are extracted in correct alphabetical order. Consider the heap of Marine officers drawn below.



The name which occurs first alphabetically is always the root. So the first step in the extraction is to remove Conklin to the final sorted list. Next, one of the leaves, specifically the rightmost child at the deepest level of the tree, replaces Conklin at the top. So the tree becomes,



and the new root, Speer, is exchanged with its smallest child until the tree is again a heap. In the tree above, Speer is exchanged with D'Arc, and then with Guernica. The resultant heap is,



The root is then extracted and the procedure is repeated until every name has been removed from the heap and is in the final list.

Problem

Given the following global types, write a heap sort for the list of Marine officers.

```
subtype Officer_Name_Type is String (1 .. 15);

type List_Type is array (Positive range <>) of Officer_Name_Type;
```

Solution and Discussion

In the implementation of the heapsort for the list of Marine officers there will be two main subprograms, one to add a name to the heap and one to extract the top name and adjust the heap. So the body of the main procedure will appear,

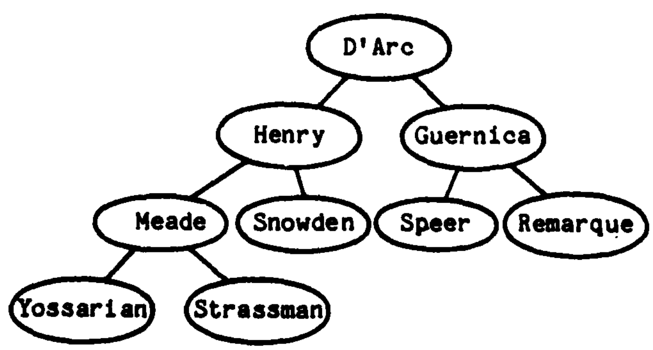
```
for Index in List'Range loop            -- Build a heap.
    Add_Name (Heap, List(Index));
end loop;

for Index in List'Range loop            -- Extract name
    Extract_Name (Heap, List(Index)); -- and rebuild heap.
end loop;
```

where List is the original array of the names. Before writing these procedures, let us consider the type declarations for the heap structure.

Heaps are often implemented using linked lists, which are naturally recommended by the tree structure. However, the use of the heap in most heapsorts (as opposed, for example, to queues) permits an array structure. Note that the only operations performed on the heap in the heapsort algorithm are the swap between a node and its child, the removal of the root, its replacement by a leaf, and the deletion of a leaf (the last element of the array in an array implementation). The usual reasons for using a linked list, varying length (the sort heap is shortened only by the deletion of the last element when it is moved to the position at the root) and frequent insertions and deletions, do not apply.

The heap is therefore defined as an array of type List_Type. The order of the array elements corresponds to descending levels of the tree. The array on the left in the diagram below represents the tree at the right:

D'Arc
Henry
Guernica
Meade
Snowden
Speer
Remarque
Yossarian
Strassman



Note that the children of a node at position n in the array are at positions 2n and 2n + 1. Consequently, all leaves are located at the end of this array.

The heapsort itself constitutes the main procedure, accepting a list as its input. As noted earlier, performing a heapsort requires performing several operations on the heap itself. For clarity, these are declared as nested subprograms within the heapsort procedure:

```
function Last_Name (Heap : List_Type) return Positive is separate;

procedure Swap (Index_1, Index_2: in Positive) is separate;

procedure Heapify (Heap : in out List_Type) is separate;

procedure Add_Name (Name : in Officer_Name_Type;
                    Heap : in out List_Type) is separate;

procedure Extract_Name (Heap : in List_Type;
                        Name : out Officer_Name_Type) is separate;
```

The procedures Add_Name and Extract_Name were mentioned at the beginning of the discussion. The implementation of these subprograms in turn uses the auxiliary procedures and function also nested inside the main program.

Add_Name is used to build the initial heap. This process involves two steps: putting the name to be added at the root of the tree, then making any necessary changes to the tree so that it remains a heap. This last step is accomplished by the procedure Heapify. The algorithm for Heapify steps through the parent nodes in the tree, swapping the parent with its smallest child. Several conditions hold for parent nodes. Because a parent located at index position n has its children located at 2n and 2n+1, it follows that the last parent node in the array is at the midpoint of the array. Thus, a node is a parent node if

```
Index <= (Heap'Last - 1)/2
```

Not all parent nodes are out of order with respect to their children. Thus, an additional check is made to determine whether the parent should be swapped with its child:

```
Heap (Index) > Heap (2 * Index) or
Heap (Index) > (Heap (2 * Index + 1))
```

Lastly, a check must be made to determine whether the assumed parent node is in fact still in the heap. The procedure Extract_Name, to be discussed shortly, removes a name from the heap, leaving a heap with n-1 valid names and one blank name. The parent node, therefore, must not contain a blank name:

```
Heap (Index)/= Blank_Name
```

If these three conditions are satisfied, then the swap is performed between the parent and its smallest child:

```
if Heap(2 * Index) < Heap(2 * Index + 1) then  -- Swap with
    Swap (Index, 2 * Index);                   -- smallest
    Index := 2 * Index;                        -- child and
else                                           -- update Index.
     Swap (Index, 2 * Index + 1);
     Index := 2 * Index + 1;
 end if;
```

Notice how the index is updated at the time of the swap. It is not incremented by 1, which would result in an algorithm that steps through each parent node of the tree, because the list which it manipulates would be a heap except for the position of this one element. (Recall that the list is incrementally transformed into a heap, *one name* at a time.) It is, therefore, sufficient in this case to pursue a single thread through the heap, namely that which "bubbles" the added element to its correct position.

The procedure Extract_Name also makes use of Heapify. The root of the heap (the first element of the array) is always the name that is first in lexicographic ordering of all the names currently on the heap. This name is removed from the heap for placement back in the list (the name is an OUT parameter of the procedure Extract_Name), and the name at

the bottom of the heap now replaces the root and is "bubbled" into position by calling Heapify:

```
procedure Extract_Name (Heap : in List_Type;
                        Name : out Officer_Name_Type) is

   Index : Positive := Last_Name (Heap);

begin  -- Extract_Name.

   Name := Heap(1);

   Heap (1)     := Heap (Index);
   Heap (Index) := Blank_Name;

   Heapify (Heap);

end Extract_Name;
```

Note that what was the last name on the heap is explicitly voided so that the heap effectively appears to decrease in size.

The complete solution code follows:

```
procedure Heapsort_Marine_Officer_List (List : in out List_Type) is

    Blank_Name : constant Officer_Name_Type := "                ";

    Heap : List_Type (List'Range) :=
             (Heap'First .. Heap'Last => Blank_Name);

    function Last_Name (Heap : List_Type) return Positive is separate;

    procedure Swap (Index_1, Index_2: in Positive) is separate;

    procedure Heapify (Heap : in out List_Type) is separate;

    procedure Add_Name (Name : in Officer_Name_Type;
                        Heap : in out List_Type) is separate;

    procedure Extract_Name (Heap : in List_Type;
                            Name : out Officer_Name_Type) is separate;

begin  -- Heapsort_Marine_Officer_List.

    for Index in List'Range loop          -- Build a heap.
        Add_Name (Heap, List(Index));
    end loop;

    for Index in List'Range loop          -- Extract names.
        Extract_Name (Heap, List(Index));
    end loop;

end Heapsort_Marine_Officer_List;


separate (Heapsort_Marine_Officer_List)
function Last_Name (Heap : List_Type) return Positive is

    Position : Positive := 1;

begin  -- Last_Name.

    while Position /= Blank_Name loop
        Position := Position + 1;
    end loop;

    return Position - 1;

end Last_Name;
```

```
separate (Heapsort_Marine_Officer_List)
procedure Swap (Index_1, Index_2 : in Positive)

      Temp : Officer_Name_Type := Heap(Index_1);

begin  -- Swap.

    Heap(Index_1) := Heap(Index_2);
    Heap(Index_2) := Temp,

end Swap;


separate (Heapsort Marine_Officer_List)
procedure Heapify (Heap : in out List_Type) is

    Index       : Positive := 1;

begin  -- Heapify.

    while                                             -- While Index is
          Index<= ((Heap'Last - 1) / 2) and           -- a parent and
               Heap(Index) /= Blank_Name and          -- within heap and
             (Heap(Index) > Heap(2 * Index) or        -- >a child,
            Heap(Index) > Heap(2 * Index + 1)) loop

        if Heap(2 * Index) < Heap(2 * Index + 1) then   -- Swap with
            Swap (Index, 2 * Index);                    -- smallest
            Index := 2 * Index;                         -- child and
        else                                            -- update Index.
            Swap (Index, 2 * Index + 1);
            Index := 2 * Index + 1;
        end if;

    end loop;

end Heapify;

separate (Heapsort_Marine_Officer_List)
procedure Add_Name (Name : in Officer_Name_Type;
                    Heap : in out List_Type) is

    End_of_Heap : Positive range Heap'Range;
```

```
begin  -- Add_Name.

    End_of_Heap := Last_Name (Heap) + 1;

    Heap (Heap'First + 1 .. End_of_Heap) :=     -- Move each name back
        Heap (Heap'First .. End_of_Heap - 1);   -- 1 position in array.

    Heap(1) := Name;                            -- Put name at root.

    Heapify (Heap);                             -- Adjust heap.

end Add_Name;


separate (Heapsort_Marine_Officer_List)
procedure Extract_Name (Heap : in List_Type;
                        Name : out Officer_Name_Type) is

    Index : Positive := Last_Name (Heap);

begin  -- Extract_Name.

    Name := Heap(1);

    Heap (1)     := Heap (Index);
    Heap (Index) := Blank_Name;

    Heapify (Heap);

end Extract_Name;
```

CHAPTER 6

ADVANCED DATA STRUCTURES

EXERCISE 6.1

SETS: GENERAL IMPLEMENTATIONS

Objective

The tutorial in Exercise 2.1 introduced the general concepts of sets. In this chapter, we illustrate certain advanced topics in set implementation, such as the use of linked lists.

Tutorial

The first section in this tutorial is a brief review of some of the relevant ideas in Exercise 2.1. Thereafter, there are several sections, each concentrating on a specific Ada implementation technique.

OPERATIONS ON SETS

For a given universe U, and sets S1 and S2 and an element E in this universe, there are certain useful operations that must be provided. These are listed below.

```
size (S1)        -- the size (i.e., the number of elements in the set)
create (S1)      -- return a set S1 given the list of elements in it
retrieve (S1)    -- return the list of elements in a set
destroy (S1)     -- destroy a given set (e.g., free up the space used by it)
complement (S1)  -- for the given universe U, return the set containing
                 -- all the elements not in S1 (equal to U - S1)

S1 + S2          -- the union (i.e., the set containing elements that appear
                 --     either in S1 or in S2 or in both)
S1 * S2          -- the intersection (i.e., the set containing elements that
                 --     appear in both S1 and S2)
S1 - S2          -- the difference (i.e., the set containing elements that
                 --     appear in S1 but not in S2)
S1 <= S2         -- the subset relationship (i.e., a boolean indicating if
                 --     all elements in S1 are in S2)
S1 = S2          -- the equality relationship (i.e., a boolean indicating if
                 --     an element is in S1 if and only if it is also in S2)
                 --     This is equivalent to S1 <= S2 and S2 <= S1.
assign (S1,S2)   -- the assignment function:  S2 := S1
```

```
insert (E,S1)   -- insert the element E into the set S1
delete (E,S1)   -- delete the element E from the set S1
member (E,S1)   -- boolean indicating if E is an element in set S1
```

BOOLEAN ARRAY IMPLEMENTATION

This representation of sets has been dealt with in detail in Exercise 2.1. Essentially, all sets are represented by Boolean arrays whose size equals the number of elements in the Universe set U, say n. Let the elements in U be e1,e2,.....,en. Then a set S will be represented by a Boolean array s1,s2,....,sn such that for i in 1 .. n,

si = True if ei is an element of S
   = False otherwise.

In particular, the Universe set U will have all values set to true, and the Empty set will have all values set to false. Thereafter, the Boolean operators "and", "or" and "not" can be used to implement all the functions indicated above. It may be noted from our definition above that the elements of all the sets are in ascending order, because it is implemented as an array indexed by a discrete type, which is itself ordered. This is convenient; in other representations, the ordering may have to be performed specifically. We shall generate here a generalized version of the implementation shown in Exercise 2.1. The concepts of data abstraction and information hiding which were introduced in Exercise 3.2 may be applied here usefully. A package is the obvious Ada structure to use. The package specification will make available a set type and the operations defined above. A user will be able to create new objects of the set type and to perform precisely the specified operations on them. This dictates that the set type be declared as a private type.

Declaring the data type as a private type would be sufficient in many cases: it is sufficient for the Boolean array representation, since we can use the predefined operations on arrays without any undesirable side effects. However, in the linked list representation, the predefined operations on pointers will cause side effects, and, therefore, we shall

provide all possible operations including assignment and equality. Thus we shall declare this type as a limited private type which has no predefined functions whatsoever. More on this issue will be mentioned later in this section. In addition, in the interests of generality, it is desirable that the package be made generic, so that different instantiations can be used to implement sets of different description. The generic formal type will be the base type of the set, and is limited to enumeration types and integer subtypes. The base type of the set is the type that defines all elements in the Universe set. Thus the generic formal will look like this:

```
generic
    type Universe_Type is (<>);
```

The visible part of the package specification may now be written. It will essentially be the same for different internal representations. It will consist of a set type called Set_Type, an unconstrained array type List_Type which will be used to specify the elements in a set of type Set_Type, and the functions that we defined above.

```
package Set_Package is

    type Set_Type is limited private;
    type List_Type is array (Natural range <>)
            of Universe_Type;

    Universe_Set, Empty_Set : constant Set_Type;
                        -- deferred constant declaration

    function Size (Set : Set_Type) return Natural;
    function Create (List : List_Type) return Set_Type;
    function Create (Element : Universe_Type) return Set_Type;
    function Retrieve (Set : Set_Type) return List_Type;
    procedure Destroy (Set : in Set_Type);
    function Complement (Set : Set_Type) return Set_Type;

    function "+" (Set1, Set2 : Set_Type) return Set_Type;
                        -- Union
    function "*" (Set1, Set2 : Set_Type) return Set_Type;
                        -- Intersection
    function "-" (Set1, Set2 : Set_Type) return Set_Type;
                        -- Difference
    function "<=" (Set1, Set2 : Set_Type) return Boolean;
                        -- Subset
    function "=" (Set1, Set2 : Set_Type) return Boolean;
                        -- Equality
    procedure Assign (Source_Set : in Set_Type;
                      Target_Set : in out Set_Type);
                        -- Assign Source to Target

    procedure Insert (Element : in Universe_Type;
                      Into_Set: in out Set_Type);
    procedure Delete (Element : in Universe_Type;
                      From_Set: in out Set_Type);
    function  Member (Element : Universe_Type;
                      Of_Set  : Set_Type)
                     return Boolean;


private
    type Set_Type is array (Universe_Type) of Boolean;

    Universe_Set : constant Set_Type :=
                        (Set_Type'Range => True);
    Empty_Set : constant Set_Type :=
                        (Set_Type'Range => False);
end Set_Package;
```

The overloading of the function Create is useful. In some cases, it would be desirable to create a new set consisting of only one element, not a list. The private portion of this package will naturally vary for different implementations of the set representation. The package body follows easily from the specification.

```
package body Set_Package is

    function Size (Set : Set_Type) return Natural is

        Count : Natural := 0;

    begin  -- Size

        for I in Set_Type'Range loop
            if Set (I) then
                 Count := Count + 1;
            end if;
        end loop;

        return Count;

    end Size;

    function Create (List : List_Type) return Set_Type is
        Set : Set_Type := (Set_Type'Range => False);

    begin  -- Create

        for I in List'Range loop
            Set (List (I)) := True;
        end loop;

        return Set;

    end Create;


    function Create (Element : Universe_Type) return Set_Type is
        Set : Set_Type := (Set_Type'Range => False);

    begin  -- Create

         Set (Element) := True;

         return Set;

    end Create;
```

```
function Retrieve (Set : Set_Type)
                   return  List_Type is
    List : List_Type (1 .. Size (Set));
    Count : Natural := 1;

begin  -- Retrieve

    for I in Set_Type'Range loop
            if Set (I) then
                List (Count) := I;
                Count := Count + 1;
            end if;
    end loop;

    return List;

end Retrieve;


procedure Destroy (Set : in Set_Type) is

begin

    null;
    -- in this representation of sets, there is no easy
    -- way of reclaiming the space used.  This function
    -- is much more relevant in the linked-list implementation.

end Destroy;


function Complement (Set : Set_Type) return Set_Type is

begin  -- Complement

    return (not (Set));

end Complement;


function "+" (Set1, Set2 : Set_Type) return Set_Type is

begin  -- Union

    return (Set1 or Set2);

end "+";       -- Union
```

```
function "*" (Set1, Set2 : Set_Type) return Set_Type is

begin  -- Intersection

    return (Set1 and Set2);

end "*";        -- Intersection

function "-" (Set1, Set2 : Set_Type) return Set_Type is

begin  -- Difference

    return (Set1 and (not (Set2)));

end "-";        -- Difference

function "<=" (Set1, Set2 : Set_Type) return Boolean is

begin  -- Subset

     return ((Set1 and Set2) = Set1);

end "<=";       -- Subset

function "=" (Set1, Set2 : Set_Type) return Boolean is

begin  -- Equality

     return ((Set1  = Set2) and (Set2  = Set1));

end "=";        -- Equality

procedure Assign (Source_Set : in Set_Type;
                  Target_Set : in out Set_Type) is

begin  -- Assign

        Target_Set := Source_Set;

end Assign;
```

```
procedure Insert (Element  : in Universe_Type;
                  Into_Set : in out Set_Type) is

begin -- Insert

        Into_Set (Element) := True;

end Insert;


procedure Delete (Element  : in Universe_Type;
                  From_Set : in out Set_Type) is

begin -- Delete

        From_Set (Element) := False;

end Delete;


function  Member (Element : Universe_Type;
                  Of_Set  : Set_Type)
                return Boolean is

begin -- Member

        return Of_Set (Element);

end Member;

end Set_Package;
```

In order to create an instance of this package, Universe_Type has to be defined first. Let us say that it is the set of highways in Boston. Thus,

```
type Boston_Highways is
    (Rte95, Rte128, Rte93, Rte9, Rte16, Rte2);
```

The city contractors entrusted with snowplowing the highways can be tracked by instantiating the Set_Package.

```
package Plowing_Set is new Set_Package (Boston_Highways);
use Plowing_Set;
```

The following declare the domains of contractors Kelly, Perini, and Lee.

```
Kelly_Co, Perini_Inc, Lee_And_Son : Set_Type;
```

To give Kelly custody of Route 9, we can create the set thus:

```
Kelly_Co := Create (Rte9);
```

To add Rte16 to Kelly's domain, we do:

```
Insert (Rte16, Kelly_Co);
```

The set that contains the routes covered by both Kelly and Perini is given by:

```
Kelly_Co * Perini_Inc
```

and the function

```
Lee_And_Son <= Perini_Inc
```

tells whether Perini's domain includes Lee's.

To see which routes are covered by Lee or Perini or both, we invoke

```
Lee_And_Son + Perini_Inc
```

and to delete Rte95 from Lee's set of roads, we execute

```
Delete (Rte95, Lee_And_Son);
```

To find out whether Perini has control over Rte2, the function to be invoked is

```
Member (Rte2, Perini_Inc)
```

which returns a Boolean. The other subprograms provided can be exercised in similar fashion.

Problem

Implement a set package providing all the functions described above, using the linked list representation.

Discussion and Solution

The general outline of the solution will be similar to that in the previous implementation: a generic package specification with a private declaration of the set type; and a package body which contains the bodies of the functions. There will be a generic formal type, the base type of the set, called Universe_Type. The data structure used to represent sets will be linked lists instead of Boolean arrays. Linked lists have been described in detail in Exercise 2.2. The type of the nodes has to be decided upon first. Clearly, the information in each node should contain at least the following: the set element, and a pointer to the next node in the list. Thus the type Node_Type can be declared as follows, with an incomplete type declaration necessary for the access type.

```
type Node_Type;
type Link is access Node_Type;
type Node_Type is
    record
        Element : Universe_Type;
        Next : Link;
    end record;
```

An interesting possibility is that the user could, instead of depending on the predefined language mechanisms, provide his/her own heap management structure. In our case, that would probably mean that a large array of type Node_Type is necessary. There would be a list of Free nodes, each of which can be allotted by using a Request_Node function (as opposed to the predefined NEW function). When nodes were no longer in use, they would be returned to the Free list. An elaborate garbage collection mechanism could also be implemented. Of course, if a user requests more nodes than are available, we would have exhausted our heap. In any case, it would be a useful exercise to implement a local storage management mechanism; however, its implementation is beyond the scope of the solution provided here.

A set will consist of a list of objects of type Node_Type, and a set will be defined by the pointer to the head of the list. As before, we can present the generic package specification as follows.

```
generic
    type Universe_Type is (<>);
package Set_Package is

    type Set_Type is limited private;
    type List_Type is array (Natural range <>)
                of Universe_Type;

    Universe_Set, Empty_Set : constant Set_Type;

    function Size (Set : Set_Type) return Natural;
    function Create (List : List_Type) return Set_Type;
    function Create (Element : Universe_Type) return Set_Type;
    function Retrieve (Set : Set_Type) return List_Type;
    procedure Destroy (Set : in Set_Type);
    function Complement (Set : Set_Type) return Set_Type;

    function "+" (Set1, Set2 : Set_Type) return Set_Type;
                    -- Union
    function "*" (Set1, Set2 : Set_Type) return Set_Type;
                    -- Intersection
    function "-" (Set1, Set2 : Set_Type) return Set_Type;
                    -- Difference
    function "<=" (Set1, Set2 : Set_Type) return Boolean;
                    -- Subset
    function "=" (Set1, Set2 : Set_Type) return Boolean;
                    -- Equality
    procedure Assign (Source_Set : in Set_Type;
                      Target_Set : in out Set_Type);
                        -- Assign Source to Target

    procedure Insert (Element : in Universe_Type;
                      Into_Set: in out Set_Type);
    procedure Delete (Element : in Universe_Type;
                      From_Set: in out Set_Type);
    function  Member (Element : Universe_Type;
                      Of_Set  : Set_Type)
                     return Boolean;
```

```
private
    type Node_Type;
    type Set_Type is access Node_Type;
    type Node_Type is
        record
            Element : Universe_Type;
            Next : Set_Type := null;
        end record;

     Universe_Set : constant Set_Type :=
                    new Node_Type'
                         (Element => Universe_Type'First,
                          Next => null);

     Empty_Set : constant Set_Type := null;

 end Set_Package;
```

The private portion of this package is different from that in the Boolean array representation. It will be noticed that the constant set Universe_Set could not be completed in the specification since it requires the allocation of several links. This can, however, be done in the executable part of the body.

The body of Set_Package can be written now. There is a design decision to be made at this point. Should the list be ordered or not? The union operation will be easily performed if it is an unordered list. However, membership and deletion will be difficult. In our design, most of the algorithms would work better on ordered lists, and therefore that is the way our lists are structured. The idea of "order" is simply the "<" relation; in other words, all our lists will have their elements in ascending order.

The size of a given set can be found by looping through the nodes of the linked list, incrementing a counter, until the end of the list.

```
function Size (Set : Set_Type) return Natural is
   Count : Natural := 1;
   Node_Ptr : Set_Type;

begin  -- Size

   if Set = null then
      return 0;
   end if;      -- null set has size zero

   Node_Ptr := Set;
   while (Node_Ptr /= null) loop
       Count := Count + 1; -- increment Count
       Node_Ptr := Node_Ptr.Next;  -- next node
   end loop;

   return Count;

end Size;
```

The function Create will create a new linked list from the input list of elements. For each element in the list, a new node is allocated and chained to the linked list. Since we require that the linked list be ordered and have no duplicates, it will be necessary to perform some additional operations to decide on the proper position of the current element. These problems could be left to the function Insert, and Create could call Insert repeatedly with each element of the input list. However, this seems a little inefficient and, therefore, we explicitly perform the necessary operations in the body of Create itself. If the input list is empty, the empty set is returned; otherwise, a new node is created which will be the head of the set to be returned. For each item in the input array, it is determined whether or not that value is already in the set: if it is, we move on to the next element in the array. Non-duplicate elements are entered into the linked list at the appropriate point in the ordering. To find this point, we loop through all the values that are lower, and as soon as we encounter a higher value, insert the element in the list just before the higher value. The local variable Set serves as the head of the output list: Prev is for backtracking to the previous node; Node_Ptr is the current node.

```
function Create (List : List_Type) return Set_Type is
    Set, Prev : Set_Type := null;
    Node_Ptr : Set_Type := null;
    Inserted : Boolean := False;

begin  -- Create

    if List'First < List'Last then
         return Empty_Set;
    end if;

    Set := new Node_Type' (Universe_Type'First, null);
    Set.Element := List (List'First);

    for I in List'First + 1 .. List'Last loop
        Node_Ptr := Set;
        Prev := Set;
        Inserted := False;

        while (Node_Ptr /= null) and not (Inserted) loop
            if Node_Ptr.Element = List (I)  then
                   Inserted := True; -- it's a duplicate
            elsif
               Node_Ptr.Element < List (I) then
                   Prev := Node_Ptr;
                   Node_Ptr := Node_Ptr.Next;
                            -- continue looping
            else
                 -- Node_Ptr.Element > List (I)
                   Node_Ptr := new Node_Type'
                                    (Element => List (I),
                                     Next => Prev.Next);
                   Prev.Next := Node_Ptr;
                   Inserted := True;
            end if;
        end loop;

    end loop;

    return Set;

end Create;
```

The overloaded function Create creates a set given one of its elements. A single node is created, and a pointer to it is returned. The node has as its value the input element.

```
function Create (Element : Universe_Type) return Set_Type is
    Set : Set_Type;

begin

    Set := new Node_Type'
              (Element =>Element,
               Next => null);

    return Set;

end Create;
```

The function Retrieve will return a list containing the elements of a given set. It is a non-destructive operation, since the set continues to exist.

```
function Retrieve (Set : Set_Type)
                   return List_Type is
    Node_Ptr : Set_Type := Set;
    List : List_Type (1 .. Size (Set));

begin  -- Retrieve

    if Size (Set) = 0 then
       return List;
    end if;             -- empty list has no elements

    for I in List'Range loop
        List (I) := Node_Ptr.Element;
        Node_Ptr := Node_Ptr.Next;
    end loop;

    return List;

end Retrieve;
```

The Destroy procedure destroys a set and releases the space used so that it may be garbage-collected and re-used. The algorithm loops through the linked list and releases each node.

```
procedure Destroy (Set : Set_Type) is
    Node_Ptr, Prev : Set_Type := Set;

begin

    while Node_Ptr /= null loop
        Prev := Node_Ptr;
        Node_Ptr := Node_Ptr.Next;
        Prev.Next := null;  -- make current node unreachable
                            -- and hence garbage collectable
    end loop;

end Destroy;
```

The union function ("+") is implemented as follows: create a new set which is equal to set S1. Then loop through set S2. If an element of S2 is already in S1, then do nothing. Otherwise, insert the element into the new set. Finally, return the new set. This means that every element in S2 was already in the old S1, or has been inserted into the new set.

The reason for creating a new set instead of performing the same operations on S1 is the following: if S1 were to be modified and returned, then there would be side-effects to the union operation. Thus the effect of executing the operation S1 + S2 would modify the value of S1 in an unexpected fashion, and this is not acceptable.

```
function "+" (Set1, Set2 : Set_Type) return Set_Type is
    Node_Ptr : Set_Type := Set2;
    New_Set : Set_Type := null;

begin  -- "+"

    Assign (Set1, New_Set);

    if Set2 = Empty_Set then
        return New_Set;
    end if;
```

```
        while (Node_Ptr /= null) loop  -- thru S2's elements
             if Member (Node_Ptr.Element, New_Set) then
                  null;       -- do nothing: it's already there
             else
                 Insert (Node_Ptr.Element, New_Set);
             end if;          -- insert it into the new set

             Node_Ptr := Node_Ptr.Next;
        end loop;

        return New_Set;           -- return enhanced New_Set

    end "+";
```

The intersection function "*" is implemented in the following fashion: if either of the input sets is empty, the intersection is empty. Otherwise, create a new list New_Set which is the same as S1. Loop through New_Set and for each element in New_Set that is not also a member of S2, delete that element from New_Set.

```
    function "*" (Set1, Set2: Set_Type) return Set_Type is
        New_Set_Ptr, Prev : Set_Type;
        New_Set : Set_Type := null;

    begin  -- "*"

        if Set1 = Empty_Set or Set2 = Empty_Set
             then  return Empty_Set;
        end if;

        Assign (Set1, New_Set);
        New_Set_Ptr := New_Set;

        while (New_Set_Ptr /= null) loop   -- thru New_Set (ie. S1)
             if Member (New_Set_Ptr.Element, Set2) then
                       -- this element is in the intersection
                 Prev := New_Set_Ptr;
                 New_Set_Ptr := New_Set_Ptr.Next;
             else
                 -- delete that element from New_Set
                 Prev.Next := New_Set_Ptr.Next;
                 New_Set_Ptr := New_Set_Ptr.Next;
             end if;
        end loop;

        return New_Set;

    end "*";
```

The function "-" for set difference S1 - S2 is implemented as follows: if S1 is the empty set, the difference is the empty set. Else create a copy of S1 in New_Set. If S2 is the empty set, the difference is New_Set (equal to S1). Otherwise, loop through the elements of New_Set. If an element of New_Set is also a member of S2, then delete it from New_Set. If it isn't, retain it. Return New_Set. New_Set is the head of the new set; New_Set_Ptr and Prev are the current and previous nodes on New_Set.

```
    function "-" (Set1, Set2 : Set_Type) return Set_Type is
        New_Set_Ptr, Prev : Set_Type;
        New_Set : Set_Type := null;

    begin  -- "-"

        if Set1 = Empty_Set then
             return Empty_Set;
        end if;

        Assign (Set1, New_Set);
        New_Set_Ptr := New_Set;

        while (New_Set_Ptr /= null) loop   -- thru New_Set (ie. S1)
             if Member (New_Set_Ptr.Element, Set2) then
                       -- member of S2, delete from New_Set by
                       -- linking previous node to next node
                  Prev.Next := New_Set_Ptr.Next;
                  New_Set_Ptr := New_Set_Ptr.Next;
             else
                       -- not a member of S2; continue looping
                  Prev := New_Set_Ptr;
                  New_Set_Ptr := New_Set_Ptr.Next;
             end if;
        end loop;

        return New_Set;

    end "-";
```

The function Is_Subset ("<=") is implemented in this fashion: the empty set is a subset of anything; and nothing is a subset of the empty set. Loop through set S1, checking if each member is also a member of S2. If so, return True; if not, False.

```
function "<=" (Set1, Set2 : Set_Type) return Boolean is
    Is_Subset : Boolean := True;
    Node_1_Ptr : Set_Type := Set1;

begin  -- "<="

     if Set1 = Empty_Set then
          return True;
     elsif Set2 = Empty_Set then
          return False;
     end if;

     while Is_Subset and (Node_1_Ptr /= null) loop
         Is_Subset := Is_Subset and
                 Member (Node_1_Ptr.Element, Set2);
         Node_1_Ptr := Node_1_Ptr.Next;
     end loop;

     return Is_Subset;

end "<=";
```

The equality function ("=") has the following rationale: if both the sets point to the same linked list, then they are equal. However, and this explains our use of limited private types, if the two set pointers do not point to the same list, that does not mean that the sets are necessarily different. Thus direct inequality, as predefined for access types, is not sufficient to guarantee unequal sets. In the algorithm, we further use the simple fact that the Empty_Set is not equal to anything but itself for some short-circuit evaluation. In the general case, we loop through both the lists until one of them terminates or they differ in some element. Since the lists are ordered, if both lists are equal, their elements will be in identical positions. If all the elements are the same and both lists have the same length, then they are equal.

```
function "=" (Set1, Set2 : Set_Type) return Boolean is
    Node_1_Ptr : Set_Type := Set1;
    Node_2_Ptr : Set_Type := Set2;

begin  -- "="

    if Set1 = Set2 then return True;  -- point to same linked
                                      -- list
    elsif
        Set1 = Empty_Set then return False;
    elsif
        Set2 = Empty_Set then return False;
    end if;

    while Node_1_Ptr /= null and Node_2_Ptr /= null
    loop
             -- loop through both the lists

        if Node_1_Ptr.Element /= Node_2_Ptr.Element then
              return False;
        end if;
             -- if an unequal element exists, lists are
             -- unequal, too; so return false

        Node_1_Ptr := Node_1_Ptr.Next;
        Node_2_Ptr := Node_2_Ptr.Next;
             -- else continue looping
     end loop;

     if Node_1_Ptr = null and Node_2_Ptr = null then
             return True;
     else   -- one list is longer than the other
             return False;
     end if;

end "=";
```

The assignment procedure takes as input two sets and assigns one to the other. We do not simply copy the pointers, since this would lead to side-effects. For instance, if S1 is assigned to S2 by changing the S2 pointer to point to the S1 list, then any changes to set S2 will affect set S1 as well, which is usually not what the user expects. This is another reason for our limited private type declaration of the Set_Type, since the predefined pointer copy operation leads to side-effects.

Instead, we provide an assignment by creating a new linked list whose elements will be identical to those in the old linked list. In this algorithm, we loop through the existing Target Set linked list, changing the values of the existing elements to those in the Source Set. If the Target Set is longer than the Source Set, the extra elements are deleted to be garbage collected. If the Source Set is longer than the Target Set, then extra elements can be added by allocating new nodes. Source_Ptr points to elements in the source list. Target_Ptr and Target_Prev point to the current and previous elements in the target list.

```
procedure Assign (Source_Set : in Set_Type;
                  Target_Set : in out Set_Type) is
    Source_Ptr : Set_Type := Source_Set;
    Target_Ptr : Set_Type := Target_Set;
    Target_Prev : Set_Type := Target_Set;

begin  -- Assign

    if Source_Ptr = Empty_Set then
         Target_Set := Empty_Set;
    else
         while Source_Ptr /= null and Target_Ptr /= null
                                                   loop
             Target_Ptr.Element := Source_Ptr.Element;
             Source_Ptr := Source_Ptr.Next;
             Target_Prev := Target_Ptr;
             Target_Ptr := Target_Ptr.Next;
         end loop;

         if Source_Ptr = null then
             Target_Prev.Next := null;
         else
             while Source_Ptr /= null loop
                 Target_Ptr := new Node_Type;
                 Target_Ptr.Element := Source_Ptr.Element;
                 Source_Ptr := Source_Ptr.Next;
                 Target_Prev.Next := Target_Ptr;
                 Target_Prev := Target_Ptr;
              end loop;
          end if;
     end if;

 end Assign;
```

The procedure Insert is implemented as follows: loop through the elements in the sets, checking the current node against the input element. If the current element is the same as the input element, then there is no need to insert it. If the element is less, then continue to loop through the set. If the element is more, then, since the list is ordered, that is the point at which the incoming element is to be inserted. If the list terminates without any elements larger than the incoming element, then allocate a new node to insert this at the tail end of the list.

```
procedure Insert (Element : in Universe_Type;
                  Into_Set : in out Set_Type) is
    Node_Ptr, Prev : Set_Type := Into_Set;
    Not_Inserted : Boolean := True;

begin  -- Insert

    while Node_Ptr /= null and Not_Inserted loop

        if Node_Ptr.Element > Element then
                -- insert the new element here
             Node_Ptr := new Node_Type' (Element => Element,
                                         Next => Prev.Next);
             Prev.Next := Node_Ptr;
             Not_Inserted := False;
        elsif
            Node_Ptr.Element < Element then
                -- keep looping
                Prev := Node_Ptr;
                Node_Ptr := Node_Ptr.Next;
        else
                -- already exists in the list
            Not_Inserted := False;
        end if;

    end loop;

    if Node_Ptr = null and Not_Inserted then
            -- extend the list with the new element
         Node_Ptr := new Node_Type' (Element => Element,
                                     Next => null);
         Prev.Next := Node_Ptr;
     end if;

end Insert;
```

The procedure Delete is implemented in a similar fashion: since the list is ordered, deletion is easy. The algorithm loops through the list, comparing each element with the element to be deleted. If the current node's element matches this, then the current node is deleted. If it is less, then looping is continued. If the current node's element is greater than the element, then the element did not exist in the set anyway.

```
procedure Delete (Element : in Universe_Type;
                  From_Set : in out Set_Type) is
    Node_Ptr, Prev : Set_Type := From_Set;
    Not_Deleted : Boolean := True;

 begin  -- Delete

     while Node_Ptr /= null and Not_Deleted loop
         if Node_Ptr.Element > Element then
                -- element wasn't in the set
              Not_Deleted := False;
         elsif
            Node_Ptr.Element < Element then
                -- keep looping
            Prev := Node_Ptr;
            Node_Ptr := Node_Ptr.Next;
         else
                -- delete the current element
            Prev.Next := Node_Ptr.Next;
            Not_Deleted := False;
         end if;
     end loop;

 end Delete;
```

The membership function works as follows: no element is a member of the empty set. Again, since lists are ordered, it is easy to decide if an element is indeed a member of a given set. Looping through the set, if the current element is equal to the element, then return true; if it is lower, then continue looping; if it is higher, the element is not in the set, so return false.

```
function Member (Element : in Universe_Type;
                 Of_Set  : in Set_Type)
                return Boolean is
    Node_Ptr : Set_Type := Of_Set;

begin  -- Member

    if Of_Set = null then
        return False;
    end if;

    while Node_Ptr /= null loop
        if  Node_Ptr.Element = Element then
                return True;
        elsif Node_Ptr.Element > Element then
                return False;
        else
            Node_Ptr := Node_Ptr.Next;
        end if;
     end loop;

     return False;

 end Member;
```

The Complement function has been placed here so that its declaration is after that of the "-" function, which it uses. We simply use the constant Universe_Set and the set difference function.

```
function Complement (Set : Set_Type) return Set_Type is

begin  -- Complement

    return (Universe_Set - Set);

end Complement;
```

The Create_Universe_Set function is called by the initialization part of the package body to define the constant Universe_Set. This has to be done in the body since it involves allocating several nodes. The algorithm essentially loops through Universe_Type, creating a linked list which has all of them as elements.

```
procedure Create_Universe_Set is
    New_Node, Prev : Set_Type := Empty_Set;

begin

    Prev := Universe_Set;

    for I in Universe_Type'Succ (Universe_Type'First)
                 .. Universe_Type'Last loop
               -- thru all but the first of Universe_Type
       New_Node := new Node_Type' (Element =  I,
                                   Next =  null);
       Prev.Next := New_Node;
       Prev := New_Node;
    end loop;

end Create_Universe_Set;
```

The linked-list representation of sets is somewhat more complex than that using Boolean arrays. However, it could mean considerable savings in space, since only the elements actually in a set are included in the nodes. Thus, at the cost of extra computing time, there will be savings in space, though bit arrays can be stored quite efficiently. The more important benefit is that the linked list is more versatile. For instance, mapping functions can be written which take in one set in our universe and return another also in the same universe. This can be done much more efficiently with linked lists. This is partly because information can be stored in the nodes. For example, say the universe is the set of naturally-occurring chemical elements, i.e., Hydrogen, Helium, Lithium, ... , Uranium. The information stored in each node could consist of valency, Group Number and so on. Thus, a function could be written which would, given a set of elements within the same group, return another set of elements which are likely to react with them, based on the information in the nodes. Furthermore, mathematical operations can be performed very easily on the elements of a given set. As an example, consider the set of integers 0 .. 100. Given any set in this universe, it would be easy to write a function which returns another set whose elements are the result of (truncating) integer division of the input set by 2, as in the function Half_Of acting on the set 6, 13, 46, 89, 96 will return 3, 6, 23, 44, 48 .

The program is presented here in its entirety, including specification and body.

```
generic
    type Universe_Type is (<>);
package Set_Package is

    type Set_Type is limited private;
    type List_Type is array (Natural range<>)
                of Universe_Type;

    Universe_Set, Empty_Set : constant Set_Type;

    function Size (Set : Set_Type) return Natural;
    function Create (List : List_Type) return Set_Type;
    function Create (Element : Universe_Type) return Set_Type;
    function Retrieve (Set : Set_Type) return List_Type;
    procedure Destroy (Set : in Set_Type);
    function Complement (Set : Set_Type) return Set_Type;

    function "+" (Set1, Set2 : Set_Type) return Set_Type;
                        -- Union
    function "*" (Set1, Set2 : Set_Type) return Set_Type;
                        -- Intersection
    function "-" (Set1, Set2 : Set_Type) return Set_Type;
                        -- Difference
    function "<=" (Set1, Set2 : Set_Type) return Boolean;
                        -- Subset
    function "=" (Set1, Set2 : Set_Type) return Boolean;
                        -- Equality
    procedure Assign (Source_Set : in Set_Type;
                      Target_Set : in out Set_Type);
                        -- Assign Source to Target

    procedure Insert (Element : in Universe_Type;
                      Into_Set: in out Set_Type);
    procedure Delete (Element : in Universe_Type;
                      From_Set: in out Set_Type);
    function  Member (Element : Universe_Type;
                      Of_Set  : Set_Type)
                     return Boolean;
```

```
private
    type Node_Type;
    type Set_Type is access Node_Type;
    type Node_Type is
        record
            Element : Universe_Type;
            Next : Set_Type := null;
        end record;

     Universe_Set : constant Set_Type :=
                     new Node_Type'
                          (Element => Universe_Type'First,
                           Next => null);

     Empty_Set : constant Set_Type := null;

 end Set_Package;
```

```
 package body Set_Package is

    function Size (Set : Set_Type) return Natural is
        Count : Natural := 1;
        Node_Ptr : Set_Type;

    begin  -- Size

        if Set = null then
           return 0;
        end if;       -- null set has size zero

        Node_Ptr := Set;
        while (Node_Ptr /= null) loop
            Count := Count + 1; -- increment Count
            Node_Ptr := Node_Ptr.Next;  -- next node
        end loop;

        return Count;

    end Size;
```

```
function Create (List : List_Type) return Set_Type is
    Set, Prev : Set_Type := null;
    Node_Ptr : Set_Type := null;
    Inserted : Boolean := False;

begin  -- Create

    if List'First < List'Last then
         return Empty_Set;
    end if;

    Set := new Node_Type' (Universe_Type'First, null);
    Set.Element := List (List'First);

    for I in List'First + 1 .. List'Last loop
        Node_Ptr := Set;
        Prev := Set;
        Inserted := False;

        while (Node_Ptr /= null) and not (Inserted) loop
            if Node_Ptr.Element = List (I) then
                   Inserted := True; -- it's a duplicate
            elsif
               Node_Ptr.Element < List (I) then
                   Prev := Node_Ptr;
                   Node_Ptr := Node_Ptr.Next;
                            -- continue looping
            else
                 -- Node_Ptr.Element > List (I)
                   Node_Ptr := new Node_Type'
                                   (Element => List (I),
                                    Next => Prev.Next);
                   Prev.Next := Node_Ptr;
                   Inserted := True;
            end if;
        end loop;

    end loop;

    return Set;

end Create;
```

```
function Create (Element : Universe_Type) return Set_Type is
    Set : Set_Type;

begin

    Set := new Node_Type'
               (Element => Element,
                Next => null);

    return Set;

end Create;


function Retrieve (Set : Set_Type)
                  return List_Type is
    Node_Ptr : Set_Type := Set;
    List : List_Type (1 .. Size (Set));

begin  -- Retrieve

    if Size (Set) = 0 then
       return List;
    end if;             -- empty list has no elements

    for I in List'Range loop
        List (I) := Node_Ptr.Element;
        Node_Ptr := Node_Ptr.Next;
    end loop;

    return List;

end Retrieve;


procedure Destroy (Set : Set_Type) is
    Node_Ptr, Prev : Set_Type := Set;

begin

    while Node_Ptr /= null loop
        Prev := Node_Ptr;
        Node_Ptr := Node_Ptr.Next;
        Prev.Next := null;  -- make current node unreachable
                            -- and hence garbage collectable
    end loop;

end Destroy;
```

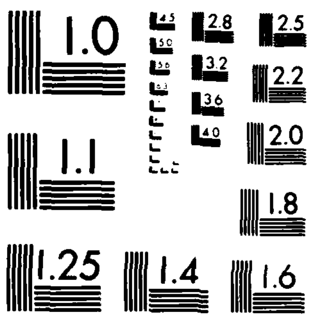
```
function "+" (Set1, Set2 : Set_Type) return Set_Type is
    Node_Ptr : Set_Type := Set2;
    New_Set : Set_Type := null;

begin  -- "+"

    Assign (Set1, New_Set);

    if Set2 = Empty_Set then
        return New_Set;
    end if;

    while (Node_Ptr /= null) loop  -- thru S2's elements
         if Member (Node_Ptr.Element, New_Set) then
              null;      -- do nothing: it's already there
         else
             Insert (Node_Ptr.Element, New_Set);
         end if;         -- insert it into the new set

         Node_Ptr := Node_Ptr.Next;
    end loop;

    return New_Set;          -- return enhanced New_Set

end "+";

function "*" (Set1, Set2 : Set_Type) return Set_Type is
    New_Set_Ptr, Prev : Set_Type;
    New_Set : Set_Type := null;

begin  -- "*"

    if Set1 = Empty_Set or Set2 = Empty_Set then
        return Empty_Set;
    end if;

    Assign (Set1, New_Set);
    New_Set_Ptr := New_Set;

    while (New_Set_Ptr /= null) loop   -- thru New_Set (ie. S1)
         if Member (New_Set_Ptr.Element, Set2) then
                   -- this element is in the intersection
             Prev := New_Set_Ptr;
             New_Set_Ptr := New_Set_Ptr.Next;
         else      -- delete that element from New_Set
             Prev.Next := New_Set_Ptr.Next;
             New_Set_Ptr := New_Set_Ptr.Next;
         end if;
    end loop;

    return New_Set;
```

AD-A146 257 ADVANCED ADA (TRADEMARK)(U) SOFTECH INC WALTHAM MA 4/5
JUL 84 DAAB07-83-C-K514

UNCLASSIFIED F/G 9/2 NL

END
DATE
FILMED
10 84
DTIC

Cont



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
function "-" (Set1, Set2 : Set_Type) return Set_Type is
    New_Set_Ptr, Prev : Set_Type;
    New_Set : Set_Type := null;

begin  -- "-"

    if Set1 = Empty_Set then
         return Empty_Set;
    end if;

    Assign (Set1, New_Set);
    New_Set_Ptr := New_Set;

    while (New_Set_Ptr /= null) loop   -- thru New_Set (ie. S1)
         if Member (New_Set_Ptr.Element, Set2) then
                -- member of S2, delete from New_Set by
                -- linking previous node to next node
             Prev.Next := New_Set_Ptr.Next;
             New_Set_Ptr := New_Set_Ptr.Next;
         else
                -- not a member of S2; continue looping
             Prev := New_Set_Ptr;
             New_Set_Ptr := New_Set_Ptr.Next;
         end if;
    end loop;

    return New_Set;

end "-";


function "<=" (Set1, Set2 : Set_Type) return Boolean is
    Is_Subset : Boolean := True;
    Node_1_Ptr : Set_Type := Set1;

begin  -- "<="

     if Set1 = Empty_Set then
          return True;
     elsif Set2 = Empty_Set then
          return False;
     end if;

     while Is_Subset and (Node_1_Ptr /= null) loop
         Is_Subset := Is_Subset and
                 Member (Node_1_Ptr.Element, Set2);
         Node_1_Ptr := Node_1_Ptr.Next;
     end loop;

     return Is_Subset;

end "<=";
```

```
function "=" (Set1, Set2 : Set_Type) return Boolean is
    Node_1_Ptr : Set_Type := Set1;
    Node_2_Ptr : Set_Type := Set2;

begin  -- "="

    if Set1 = Set2 then return True;  -- point to same linked
                                      -- list
    elsif
        Set1 = Empty_Set then return False;
    elsif
        Set2 = Empty_Set then return False;
    end if;

    while Node_1_Ptr /= null and Node_2_Ptr /= null
    loop
              -- loop through both the lists

        if Node_1_Ptr.Element /= Node_2_Ptr.Element then
            return False;
        end if;
              -- if an unequal element exists, lists are
              -- unequal, too; so return false

        Node_1_Ptr := Node_1_Ptr.Next;
        Node_2_Ptr := Node_2_Ptr.Next;
              -- else continue looping
     end loop;

     if Node_1_Ptr = null and Node_2_Ptr = null then
         return True;
     else   -- one list is longer than the other
         return False;
     end if;

end "=";
```

```
procedure Assign (Source_Set : in Set_Type;
                  Target_Set : in out Set_Type) is
    Source_Ptr : Set_Type := Source_Set;
    Target_Ptr : Set_Type := Target_Set;
    Target_Prev : Set_Type := Target_Set;

begin  -- Assign

    if Source_Ptr = Empty_Set then
        Target_Set := Empty_Set;
    else
        while Source_Ptr /= null and Target_Ptr /= null loop
            Target_Ptr.Element := Source_Ptr.Element;
            Source_Ptr := Source_Ptr.Next;
            Target_Prev := Target_Ptr;
            Target_Ptr := Target_Ptr.Next;
        end loop;

        if Source_Ptr = null then
            Target_Prev. Next := null;
        else
            while Source_Ptr /= null loop
                Target_Ptr := new Node_Type;
                Target_Ptr.Element := Source_Ptr.Element;
                Source_Ptr := Source_Ptr.Next;
                Target_Prev.Next := Target_Ptr;
                Target_Prev := Target_Ptr;
            end loop;
        end if;
    end if;

end Assign;
```

```
procedure Insert (Element : in Universe_Type;
                  Into_Set : in out Set_Type) is
    Node_Ptr, Prev : Set_Type := Into_Set;
    Not_Inserted : Boolean := True;

begin  -- Insert

    while Node_Ptr /= null and Not_Inserted loop

        if Node_Ptr.Element > Element then
              -- insert the new element here
            Node_Ptr := new Node_Type' (Element => Element,
                                        Next => Prev.Next);
            Prev.Next := Node_Ptr;
            Not_Inserted := False;
        elsif
            Node_Ptr.Element < Element then
                -- keep looping
                Prev := Node_Ptr;
                Node_Ptr := Node_Ptr.Next;
        else
                -- already exists in the list
            Not_Inserted := False;
        end if;

    end loop;

    if Node_Ptr = null and Not_Inserted then
            -- extend the list with the new element
        Node_Ptr := new Node_Type' (Element => Element,
                                    Next => null);
        Prev.Next := Node_Ptr;
    end if;

end Insert;
```

```
procedure Delete (Element : in Universe_Type;
                  From_Set : in out Set_Type) is
    Node_Ptr, Prev : Set_Type := From_Set;
    Not_Deleted : Boolean := True;

 begin  -- Delete

     while Node_Ptr /= null and Not_Deleted loop
         if Node_Ptr.Element > Element then
                 -- element wasn't in the set
              Not_Deleted := False;
         elsif
            Node_Ptr.Element < Element then
                 -- keep looping
            Prev := Node_Ptr;
            Node_Ptr := Node_Ptr.Next;
         else
                 -- delete the current element
            Prev.Next := Node_Ptr.Next;
            Not_Deleted := False;
         end if;
      end loop;

 end Delete;


 function Member (Element : in Universe_Type;
                  Of_Set  : in Set_Type)
                  return Boolean is
     Node_Ptr : Set_Type := Of_Set;

 begin  -- Member

     if Of_Set = null then
         return False;
     end if;

     while Node_Ptr /= null loop
         if  Node_Ptr.Element = Element then
                 return True;
         elsif Node_Ptr.Element > Element then
                 return False;
         else
             Node_Ptr := Node_Ptr.Next;
         end if;
      end loop;

      return False;

  end Member;
```

```
function Complement (Set : Set_Type) return Set_Type is

begin  -- Complement

    return (Universe_Set - Set);

end Complement;


procedure Create_Universe_Set is
    New_Node, Prev : Set_Type := Empty_Set;

begin

    Prev := Universe_Set;

    for I in Universe_Type'Succ (Universe_Type'First)
                 .. Universe_Type'Last loop
              -- thru all but the first of Universe_Type
       New_Node := new Node_Type' (Element => I,
                                    Next => null);
       Prev.Next := New_Node;
       Prev := New_Node;
    end loop;

end Create_Universe_Set;

begin    -- body of the package; for initialization

      Create_Universe_Set;

end Set_Package;
```

EXERCISE 6.2 GRAPHS:

REPRESENTATIONS, DIFFERENT IMPLEMENTATIONS, AND APPLICATIONS

Objective

This exercise introduces graphs, both theory and application, and it discusses several possible Ada implementations.

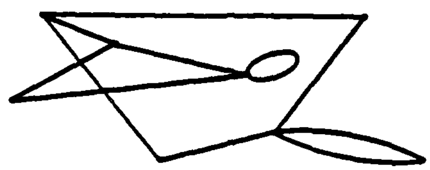
Tutorial

This tutorial consists of two parts, a theoretical section and an Ada section. The first part discusses elementary graphs, their properties and some basic algorithms. The second section shows possible implementations of graphs and graph manipulations. It should be noted that the first section uses traditional mathematical notations; any resemblance with the notation of Ada or any other programming languages is accidental.

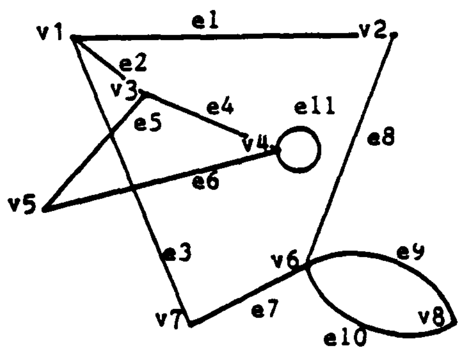
GRAPH THEORY

The graph theory which is the subject of this Tutorial does not refer to the plots of y versus x on a Cartesian coordinate system. In the context of this exercise, graphs refer to a network of points and the lines drawn to connect them. The points could represent a set of cities and the lines could be the air routes which connect them. Alternatively, the points could depict both a set of concepts and a set of textbooks, with the lines mapping concepts explained in a particular textbook. Graphs can be used for PERT (Program Evaluation Review Technique) diagrams, for management analysis, for resource allocation, for flow analysis, to name but a few areas of application. For instance a graph could model the resources available for a project and the milestones to be met; manipulating the graph, e.g. connecting resources to milestones in different ways would allow a manager to plan the most time and cost effective use of his personnel and to determine what deadlines or tasks were feasible given the current resource level.

A special terminology is used in discussing graphs. The "points" of the previous paragraph are called vertices, and the lines are known as edges. Weights may be associated with edges, indicating for instance the distance between two cities or the time needed to execute a task. The following network represents a graph:



As drawn above, it is difficult to discuss this graph because the drawing does not provide a way to disambiguate between the vertices and edges. Although graph theory itself does not require labeling of vertices and edges, most graphs generally are labeled for ease of reference. Here is the same graph with an arbitrary labeling of its vertices and edges.

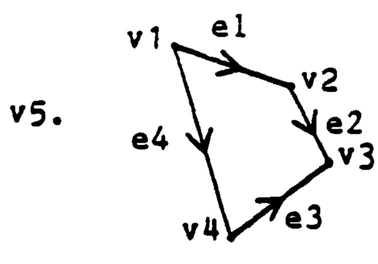


The edges may be curved lines just as well as straight lines, as in the loop at v4. Multiple edges are also allowed, as between v6 and v8. No significance is attached to whether or not edges cross each other at non-vertex intersections; what is important is which edges join at which vertices.

Edges may be identified either through a labeling as shown in the above diagram, or through the vertices which delimit the edge, as listed in the table below:

edge	vertex pair
e1	(v1, v2)
e2	(v1, v3)
e3	(v1, v7)
e4	(v3, v4)
e5	(v3, v5)
e6	(v4, v5)
e7	(v6, v7)
e8	(v2, v6)
e9	(v6, v8)
e10	(v6, v8)
e11	(v4, v4)

An edge is incident to a vertex if that vertex is one of the endpoints of that edge. The vertex v1 has three edges incident to it, e1, e2, and e3. The degree of a vertex is the number of edges incident to it, so the degree of v1 is 3. In the case of a loop, the edge is counted twice, making the degree of v4 equal to 4. The above graph illustrates an undirected graph, that is, a graph in which one can move along a given edge in either direction. In a directed graph, this freedom is restricted, and motion is only allowed in the direction specified by an arrow along the edge. In the graph below, it is possible to go from v1 to v2, but not vice versa.



Directed graphs are used to model dynamic situations such as PERT charts and flowcharts, whereas undirected graphs are used in static situations such as a database analysis.

A useful concept in moving along the edges of a graph is that of a path. Specifically, a path is an alternating sequence of vertices and edges such that no vertices (and therefore no edges) are repeated. A cycle is a special kind of path in which a single vertex is repeated because the first vertex and the last vertex are identical. The length of a path is the number of edges it contains.

The sequences e1-e8-e9, e7-e3-e2-e5-e6 and e2-e4 are examples of paths in the undirected graph drawn earlier. This graph contains two cycles, e1-e8-e7-e3 and e5-e4-e6. In the directed graph above, there are no cycles. The sequences e1-e2 and e4-e3 are legitimate paths, whereas e1-e2-e3 is not. Furthermore, notice that no path includes the vertex v5 because it is not connected to the rest of the graph, i.e., it has degree 0.

The idea of weights can be applied to both the types of graphs that were discussed above. Each edge in a graph may have associated with it a weight or a cost, which may be considered to be the cost of traversing that edge. Weighted graphs provide a useful abstraction of many real-life relationships, for instance, cities connected by highways. In this case, the vertices would represent the cities themselves, and the weights on the edges connecting them, the distances between them. The well-known "Traveling Salesman" problem, where a salesman would like to find a minimum cost tour of N cities while visiting each of them once and only once, is an example where weighted graphs are the most natural data structures to use. It may be noted that if there does not exist an edge between two vertices x and y, it would be equivalent to maintaining that there is an edge of infinite cost between them. Trees, which were discussed in Exercise 4.2, are, in fact, a special kind of graph with the following properties. They are undirected, they have no cycles, and they are connected. A connected graph is one in which for any two vertices, there is a path that joins them. Of the two examples above, only the first one shows a connected graph.

A spanning tree of a particular graph is a tree which contains all the vertices of this graph. A graph may contain more than one spanning tree. Spanning trees are useful when applying searching techniques to graphs because they limit the number of edges which must be searched. An area of application is deriving a set of equations for an electrical network. Minimum cost spanning trees are interesting because they allow one to choose an option of least cost. For instance, in building a communications network, a spanning tree can be used to find the least cost network connecting all points of interest.

The basic algorithm to derive a spanning tree consists of the following steps. Initially, all the edges of the graph must be examined and the tree is empty. Through each iteration, one examines a different edge. If this particular edge does not create a cycle in the edges constituting the spanning tree, it is added to the tree otherwise it is discarded. When all edges have been examined, then th tree is complete.

A minimum cost spanning tree can be achieved th gh a slight modification of this algorithm. Instead of choosing t edge for examination in a random order, the edges are chosen in order of increasing cost. Typically, the edges are sorted in this order before they are used in this algorithm. Another interesting problem which uses a related algorithm is that of finding the shortest path between two given vertices x and y. This naturally applies only to weighted graphs, and the objective is to find the path that connects x and y and has the lowest cost associated with it.

ADA IMPLEMENTATION OF GRAPHS

There are several possible implementations of graphs in Ada, some relying on matrices and others on linked lists. Matrices may be used to describe several properties of graphs:

incidence of a vertex v(i) to an edge e(j)
vertex adjacency (whether 2 vertices form an edge)
edge adjacency (whether 2 edges share a vertex)
spanning trees

An incidence matrix is a matrix whose rows are the vertices and whose columns are the edges. An element of this matrix has the value One if a vertex v(i) is incident to the edge e(j); it has the value Neither if the vertex v(i) is not incident to the edge e(j); and it has the value Both if the edge e(j) is a loop. For the undirected graph shown above, the declarations below create an incidence matrix type.

```
type Vertex_Type is (v1, v2, v3, v4, v5, v6, v7, v8);
type Edge_Type is
     (e1, e2, e3, e4, e5, e6, e7, e8, e9, e10, e11);
type Incidence_Type is (Neither, One, Both);
type Incidence_Matrix_Type is
        array (Vertex_Type, Edge_Type) of Incidence_Type;
Graph_1 : Incidence_Matrix_Type :=
    (v1 => ( e1 e2 e3 => One, others => Neither),
    (v2 => (    e1 e8 => One, others => Neither),
    (v3 => ( e2:e4 e5 => One, others => Neither),
    (v4 => (    e4 e6 => One, e11 => Both, others => Neither),
    (v5 => (    e5 e6 => One, others => Neither),
    (v6 => ( e7:e8 e9:e10 => One, others => Neither),
    (v7 => (    e3 e7 => One, others => Neither),
    (v8 => (    e9 e10 => One, others => Neither));
```

For a directed graph, there are four possibilities for the edges: an edge is not incident to a vertex, an edge emanates from a vertex, an edge goes to a vertex, and an edge both emanates from and goes to a vertex (the loop). These possibilities are represented through the following enumeration type:

```
type Incidence_Type is (Neither, From, To, Both);
```

The remaining type declarations needed to describe the directed graph above are:

```
type Vertex_Type is range 1 .. 5;
type Edge_Type is range 1 .. 4;
type Incidence_Matrix_Type is
        array (Vertex_Type, Edge_Type) of Incidence_Type;
Graph_2 : Incidence_Matrix_Type :=
    ((From,    Neither, Neither, From),
     (To,      From,    Neither, Neither),
     (Neither, To,      To,      Neither),
     (Neither, Neither, From,    To),
     (Neither, Neither, Neither, Neither));
```

Writing individual declarations for every graph becomes rather cumbersome. It makes sense to declare a generic package to describe a particular kind of matrix representation for a directed or undirected graph, using instantiations of it to describe a specific graph.

The next matrix representation to be discussed is the vertex adjacency matrix. This matrix is indexed by a vertex pair. An element of the matrix indicates whether for two vertices, there exists 0 or more edges connecting them. The vertices can be represented by any discrete type, either an integer type as above or an enumeration type.

```
generic

   type Vertex_Type is (<>);

package Vertex_Adjacency_Matrix_Package is

   type Vertex_Adjacency_Matrix_Type is
           array (Vertex_Type, Vertex_Type) of Natural;

   . . .    -- procedures and functions that operate on graphs
            --  and compute properties such as the degree of
            --  a vertex.

end Vertex_Adjacency_Matrix_Package;
```

The difference between an instantiation of this package for a directed graph and one for an undirected graph is that the latter always yields a symmetric matrix. For a directed graph, a matrix element stores whether there exists an edge from the vertex indicated by the row index to the vertex indicated by the column index.

```
type Graph_1_Size is Integer range 1 .. 8;
type Graph_2_Size is Integer range 1 .. 5;
package Graph_1_Package is new
   Vertex_Adjacency_Matrix_Package
      (Vertex_Type => Graph_1_Size);
package Graph_2_Package is new
   Vertex_Adjacency_Matrix_Package
      (Vertex_Type => Graph_2_Size);
```

```
Graph_1 : Graph_1_Package.Vertex_Adjacency_Matrix_Type :=
                ((0, 1, 1, 0, 0, 0, 1, 0),
                 (1, 0, 0, 0, 0, 1, 0, 0),
                 (1, 0, 0, 1, 1, 0, 0, 0),
                 (0, 0, 1, 1, 1, 0, 0, 0),
                 (0, 0, 1, 1, 0, 0, 0, 0),
                 (0, 1, 0, 0, 0, 0, 1, 2),
                 (1, 0. 0, 0, 0, 1, 0, 0),
                 (0, 0, 0, 0, 0, 2, 0, 0));

Graph_2 : Graph_2_Package.Vertex_Adjacency_Matrix_Type :=
                ((0, 1, 0, 1, 0),
                 (0, 0, 1, 0, 0),
                 (0, 0, 0, 0, 0),
                 (0, 0, 1, 0, 0),
                 (0, 0, 0, 0, 0));
```

Either the incidence or the vertex adjacency matrices alone completely describe a graph. The remaining two representations discussed show ways of representing particular features of graphs. An edge adjacency matrix is similar to a vertex adjacency matrix, except that here, the rows and columns of the matrix represent the edges of the graph, and an element of the matrix indicates whether the two indexed edges share a common vertex. This representation is used for undirected graphs.

```
generic

   type Edge_Type is (<>);

package Edge_Adjacency_Matrix_Package is

   type Edge_Adjacency_Matrix_Type is
           array (Edge_Type, Edge_Type) of Boolean;

   . . .   -- operations on graphs

end Edge_Adjacency_Matrix_Package;
```

For the undirected graph discussed earlier, an instantiation that creates an 11 x 11 matrix is needed:

```
type Graph_1_Edge_Type is range 1 .. 11;
package Graph_1_Package is new
   Edge_Adjacency_Matrix_Package (Graph_1_Edge_Type);
```

The matrix can then be initialized as follows. True and False are renamed to T and F for legibility.

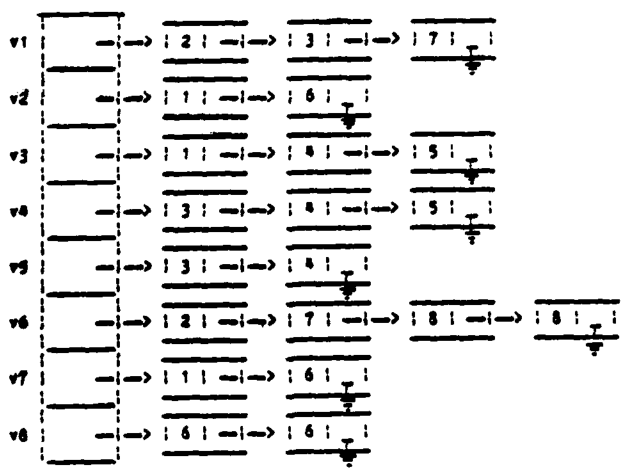
```
T : Boolean renames True;
F : Boolean renames False;

Graph_1 :=
   ((T, T, T, F, F, F, F, T, F, F, F),
    (T, T, T, T, T, F, F, F, F, F, F),
    (T, T, T, F, F, F, T, F, F, F, F),
    (F, T, F, T, T, T, F, F, F, F, T),
    (F, T, F, T, T, T, F, F, F, F, F),
    (F, F, F, T, T, T, F, F, F, F, T),
    (F, F, T, F, F, F, T, T, T, T, F),
    (T, F, F, F, F, F, T, T, T, T, F),
    (F, F, F, F, F, F, T, T, T, T, F),
    (F, F, F, F, F, F, T, T, T, T, F),
    (F, F, F, T, F, T, F, F, F, F, T));
```

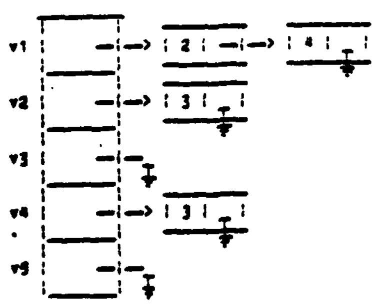
The spanning tree representation consists of a matrix whose rows are indexed by spanning trees and whose columns are indexed by edges. In order to implement this matrix, the number of spanning trees must be computed in order to determine the bounds of the matrix. A labeling of these trees is assumed in order to know how to fill the matrix. Because an edge either is or is not in a spanning tree, a matrix of Booleans as described for the edge adjacency matrix is sufficient.

Linked lists may also be used to represent graphs. In general, the matrices used to represent graphs are sparse, and for a large graph, this imposes a large storage requirement on a system. From this point of view, a linked list implementation offers considerable advantages. The most common form of linked list used for graphs is an adjacency list derived from the vertex adjacency matrix. Each vertex in the graph maintains a list of those vertices to which it is connected by an edge. Essentially, each of these lists summarizes in linked list form the information stored in the corresponding row of the adjacency matrix. The header nodes, representing the individual vertices may be stored in an array, so effectively, the data structure looks like an array of lists:

```
generic

    type Vertex_Type is (<>);

package Adjacency_List_Package is

    type Vertex_Node_Type;
    type Vertex_Pointer_Type is access Vertex_Node_Type;
    type Vertex_Node_Type is
       record
          Vertex : Vertex_Type;
          Next_Vertex : Vertex_Pointer_Type;
       end record;
    type Vertex_List_Type is
            array (Vertex_Type) of Vertex_Pointer_Type;

    . . .   -- operations on graphs

end Adjacency_List_Package;
```

A schematic representation of the graphs illustrated earlier follows. In the case of a directed graph, the nodes in the list for a given vertex indicate those nodes to which one can get, traveling from the initial vertex.



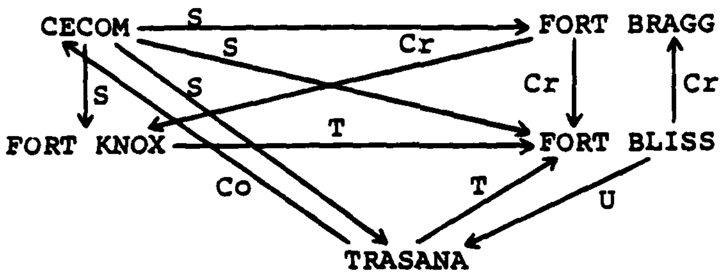


It may be noted that the same structure can be adopted for weighted graphs as well. Each of the links will contain, in addition to the name of the vertex that it is associated with, the weight of the corresponding edge. For weighted graphs a matrix representation could be used, where each matrix element specifies the cost associated with a particular edge. Because edges do not necessarily exist between every vertex pair, then the matrix element could be of a variant record type. The discriminant, a Natural number, would indicate the number of edges. The cost component would be an array whose length was constrained by the discriminant and whose elements were the costs associated with these edges.

```
type Cost_Type is ....;
type Cost_Array_Type is
        array (Positive range <>) of Cost_Type;
type Matrix_Element_Type
                (Edge_Count : Natural := 0) is
    record
        Cost_Array : Cost_Array_Type (1 .. Edge_Count);
    end record;
```

Problem

A set of Army facilities is building a telecommunications network to handle various classifications (unclassified, confidential, secret, top secret, and crypto) of communications between them. The facilities have agreed to contribute from their own budgets sufficient funding to install the outgoing links they deem necessary. No facility will install multiple links between it and another node. Assume that each link is unidirectional and that each link can transmit information only up to its specified classification. The following diagram illustrates the current plans for this network:



Develop a data structure to represent this network.

Discussion and Solution

The scenario described in the problem statement suggests a graph based solution. The diagram consists of a set of vertices, the Army facilities, which are connected by a set of edges, the communication links. Because the links are unidirectional this graph is a directed graph: a link from CECOM to Fort Bragg does not imply the existence of a link from Fort Bragg to CECOM. Thus it is important to establish the "from" and "to" vertices. Lastly, this graph is a weighted graph, where the weights are the classifications of the links.

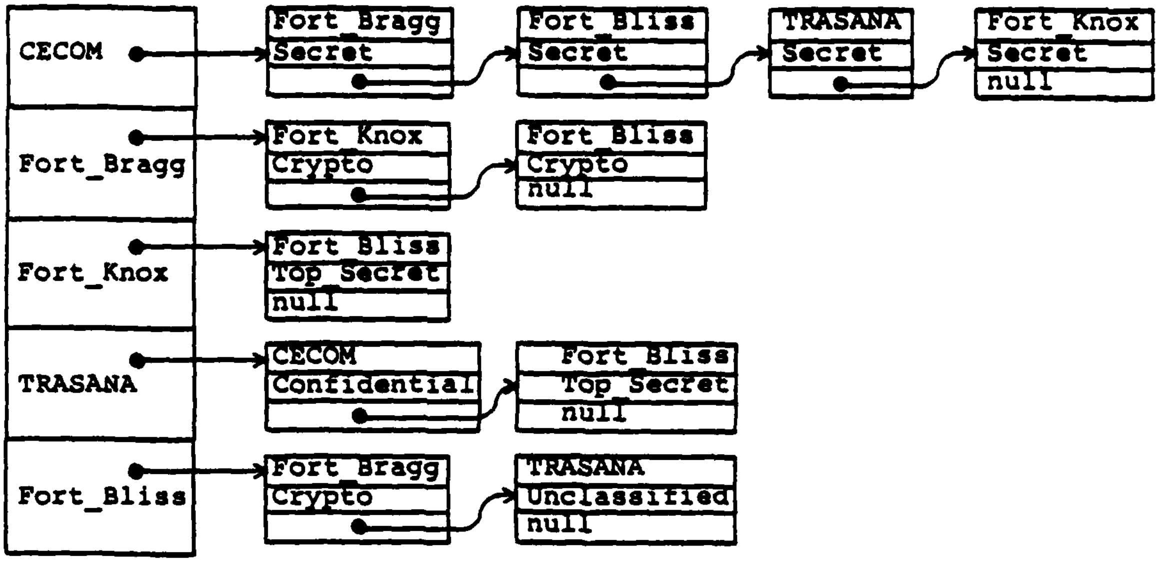
There are two kinds of data structures that would be appropriate for this graph: linked list and matrix, each of which will be discussed. The linked list representation is very similar to the generic Adjacency_List_Package depicted in the Tutorial for unweighted graphs. In addition to the Vertex_Type, a record type is needed to store the classification of the link to some destination. The supporting types are defined as:

```
type Station_Type is
      (CECOM, Fort_Bragg, Fort_Knox, TRASANA, Fort_Bliss);

type Classification_Type is
      (Unclassified, Confidential, Secret, Top_Secret, Crypto);

type Node_Type;
type Link_Type is access Node_Type;
type Node_Type is
    record
        To_Station     : Station_Type;
        Classification : Classification_Type;
        Next           : Link_Type;
    end record;
```

An array of the linked list of destination stations is now declared:

```
type Station_Links_Type is array (Station_Type) of Link_Type;
```

Pictorially, we can now create the following:



To actually create code the graph, we would declare the object,

```
Army_Network :  Station_Links_Type;
```

and use linked list manipulations to fill it with actual data:

```
Army_Network (CECOM) := new Node_Type'
                             (To_Station     => Fort_Bragg,
                              Classification => Secret,
                              Next           =>
                            new Node_Type'
                              (To_Station     => Fort_Bliss,
                               Classification => Secret,
                               Next           =>
                             new Node_Type'
                               (To_Station     => TRASANA,
                                Classification => Secret,
                                Next           =>
                              new Node_Type'
                                (To_Station     => Fort_Knox,
                                 Classification => Secret,
                                 Next           => null))));

Army_Network (Fort_Bragg)  := new  Node_Type' ( ... );
```

An alternate and equally valid implementation of graphs uses a matrix representation. The final choice of representation depends on the algorithms to be used with the graph and on the data structures on which these algorithms operate.

Each element in the matrix representation of a weighted directed graph contains the weight of the edge connecting the vertex indicated by the row index to the vertex indicated by the column index. If no edge exists, an infinite weight value is assumed. The generic packages discussed in the Tutorial assume unweighted graphs, so their declarations must be modified as shown below. The set of vertices is:

```
type Station_Type is
    (CECOM, Fort_Bragg, Fort_Bliss, TRASANA, Fort_Knox);
```

The weights ascribed to edges are the security classifications associated with each link. Observe the placement of the additional enumeration value No_Link, representing the infinite weight quantity, at the end of the list to ensure that its weight is greater than any of the other weights. Also note the classification Self, to indicate the security of a path of length 0, i.e. from one node back to itself:

```
type Classification_Type is
    (Self, Unclassified, Confidential, Secret, Top_Secret,
     Crypto, No_Link);
```

The matrix can now be declared and initialized:

```
type Station_Links_Type is array
        (Station_Type, Station_Type) of Classification_Type;
Network : Station_Links_Type :=

-----------------------------------------------------------------------
--   To:       CECOM,          Fort      Fort       TRASANA     Fort
-- From:                        Bragg,    Bliss,                  Knox
-----------------------------------------------------------------------
   (CECOM      = > (Self,      Secret,   Secret,    Secret,     Secret),
    Fort Bragg = > (No_Link, Self,       Crypto,    No_Link,    Crypto),
    Fort Bliss = > (No_Link, Crypto,     Self, Unclassified,    No_Link),
    TRASANA    = > (Confidential, No_Link, Top_Secret, Self, No_Link),
    Fort Knox  = > (No_Link, No_Link, Top_Secret, No_Link, Self));
```

In table form, the matrix looks like:

To / From	CECOM	Fort Bragg	Fort Bliss	TRASANA	Fort Knox
CECOM	self	secret	secret	secret	secret
Fort Bragg	x	self	crypto	x	crypto
Fort Bliss	x	crypto	self	unclassified	x
TRASANA	confidential	x	top secret	self	x
Fort Knox	x	x	top secret	x	self

CHAPTER 7

IMPLEMENTATION-DEPENDENT FEATURES

EXERCISE 7.1

REPRESENTATION CLAUSES

Objective

To introduce the implementation-dependent feature representation clauses.

Tutorial

The Ada language defines some features which allow a programmer to specify the physical representation of an entity, i.e., map the abstract program entity to physical hardware. These features are implementation-dependent: an implementation is not required to support these features.

The typical student of Ada will wonder, "Why would a high level language such as Ada include features which allow the underlying representation to be specified?" The reason is simple. Ada is designed to be a language for embedded systems. Often the host for these systems will be small microprocessors whose hardware configuration place requirements on the software. The embedded system software must be able to adapt to hardware requirements explicitly.

The most common uses for representation clauses are in interfacing with physical devices and in specifying the precise layout of data structures.

There are four features which allow the programmer to specify the actual representation of a program entity. We refer to these features collectively as representation clauses. They are the address clause, the length clause, the enumeration representation clause, and the record representation clause.

The address clause, is used to assign an actual memory location as the address of a specific entity in the program. For instance, it could be used in a sonar system where a sonar signal is sent out into the ocean via a sound wave. When the sound wave returns, the changes in the wave

formation are analyzed. This analysis is used to determine the size, shape, and location of objects in the sonar path. This sound wave is sent via a specific hardware device within the sonar equipment. Some mechanism must exist which transforms the internal signal into a sound wave and transforms the received sound wave into an internal signal. In others words, the program must be able to interface directly with the hardware.

Abstractly, we could think of this signal port as a file, but this does not really model the actual world because information is not being stored. Something is being done to the signal. A more realistic way of representing this process is in terms of functions and procedures which accept the signal, process it, and return the modified signal. What we really want to do is associate some object within our program with the actual signal port. The address clause was defined to handle such cases.

The syntax of the address clause is:

```
for Entity_Name use at Simple_Expression;
```

where Entity_Name is the name of an object, subprogram, package, task, or single entry of a task family; and where Simple_Expression is of the predefined type Address. (Address is defined in the predefined package System to be implementation defined.) For the purposes of this discussion we will define Address as a number which can be represented in eight bits.

```
type Address is 0 .. 255;
```

Notice that all values of this type are non-negative. A negative address simply does not make sense.

Now using the address clause, we can represent the sonar problem. Assume that in the physical system, the sonar port has address 8#12#. The software could appear as:

```
package Signal_Processor is

     type Signal_Type is ...;

     function Return_Signal return Signal_Type;
     procedure Send_Signal (Output_Signal : in Signal_Type);

end Signal_Processor;

with System;
package body Signal_Processor is

     Signal_Port_Address : constant System.Address := 8#12#;
     Signal_Port : Signal_Type;
     for Signal_Port use at Signal_Port_Address;

     procedure Send_Signal (Output_Signal : in Signal_Type) is

     begin -- Send_Signal

        ...      -- Possibly some processing
        Signal_Port := Output_Signal;

     end Send_Signal;

     function Return_Signal return Signal_Type is

     begin -- Return_Signal

        ...     -- Possibly some processing
        return Signal_Port;

     end Return_Signal;

end Signal_Processor;
```

The length clause is used to specify the amount of storage the implementation is to allow for objects of the specified type. It overrides the system default for the amount of storage an object normally receives. For instance, in a system which is tight on storage space to guarantee that the objects declared use as little space as possible, the length clause is used.

The syntax of the length clause is:

```
for Attribute use Simple_Expression;
```

where Attribute takes the form T'Attribute_Name in which T is the name of some type or subtype and Attribute_Name is Size, Storage_Size, or Small.

When Attribute has the form T'Size, T must have static constraints, and the Simple_Expression must be a static value of some integer type. This value represents the number of bits to be allocated for objects of type T and it must allow for every allowable value of these objects. For example, to guarantee that all objects of the following type,

```
type Small_Int is range 0 .. 15;
```

take as little space as possible, write the following representation clause,

```
for Small_Int use 4;
```

Note that the following:

```
type Int is range 1 .. 100;
for Int'Size use 4;         -- ILLEGAL
```

is illegal because 4 bits is not enough space to represent all possible values of objects of type Int.

When Attribute has the form T'Storage_Size, T must be either an access type or a task type (beyond the scope of this workbook). If it is an access type, Simple_Expression represents the number of storage units reserved for all objects designated by values of the access type. If the type is a task type, Simple_Expression represents the number of storage units to be reserved for an activation of a task. For both forms the value of Simple_Expression must be of some integer type. The value need not be static.

When Attribute has the form T'Small, the representation clause is used to specify the smallest positive value available for a fixed point type. T, therefore, must be the name of some fixed point type. Simple_Expression is used as the value of Small for the representation of values of the fixed point base type. It must be a static expression of some real type and its value must not be greater than the delta of the named subtype.

The enumeration representation clause is used to map enumeration literals to specific internal representations. For example, suppose an Ada program needs to represent and use the actual machine op-codes for some application. At an abstract level, we would like to refer to the op-codes in the program by their mnemonic name. We must, therefore, have some facility to directly map the mnemonic name to the actual value of the op-code.

In Ada, this is done using an enumeration type to represent the op-codes and an enumeration representation clause to map the literals to the actual values.

The syntax for the enumeration representation clause is:

```
for Type_Simple_Name use Aggregate;
```

where Type_Simple_Name is the name of an enumeration type and Aggregate is a one-dimensional array aggregate in which the enumeration type is the index subtype. The values contained in the aggregate must be of an integer type, specified by an integer literal or a named number. Every enumeration literal of the specified enumeration type must be supplied with a value in the aggregate.

Now we can represent op-codes in Ada.

```
package M6800_Op_Codes is
     type Op_Codes is (NOP, TAP, TPA, INX, DEX, CLV, ..., LDX, STX);

     for Op_Codes use (NOP => 16#01#,
                       TAP => 16#06#,
                       TPA => 16#07#,
                       INX => 16#08#,
                       DEX => 16#09#,
                       CLV => 16#0A#,
                          ...,
                       LDX => 16#FE#,
                       STX => 16#FF#);
end M6800_Op_Codes;
```

Note that in this package the predefined order relationship of the enumeration literals is maintained in the representation clause. This is required for legal enumeration representation clauses. For example,

```
for Op_Codes use (NOP => 16#14#,     -- ILLEGAL
                  TAP => 16#06#,
                  TPA => 16#07#,
                  INX => 16#08#,
                  DEX => 16#09#,
                  CLV => 16#0A#,
                  ...,
                  LDX => 16#FE#,
                  STX => 16#FF#);
```

is illegal because the order relationship of the literals is not maintained.

Note that the attributes Succ, Pred, and Pos are available, even if the values are not contiguous. Note also that the value returned by the operation Pos is not affected by the representation clause. For example,

```
Op_Codes'Pos (TPA)
```

is still 2 after the representation clause.

The record representation clause is used to specify the internal layout of a record structure, and to specify whole record alignment. For example, the order, position and size of the record's components can be specified by the representation clause.

The syntax for a record representation clause is:

```
for Type_Simple_Name use
     record [Alignment_Clause]
          {Component_Clause}
     end record;
```

The Type_Simple_Name must be the name of a record type. The alignment clause specifies the alignment of the object by specifying that the storage for each record object declared of this type be allocated at a starting address that is a multiple of the specified value.

The syntax for an alignment clause is:

```
at mod Static_Simple_Expression;
```

where Static_Simple_Expression must be a static integer value.

The Component_Clause specifies the component's storage location relative to the beginning of the record object. Its syntax is:

Component_Simple_Name at Static_Simple_Expression range Static_Range;

Again, Static_Simple_Expression must be an integer value determinable at compile time. Component_Simple_Name must be the name of a component of the record type. The Static_Range defines the bit positions for the storage of that component relative to the start of the whole records storage position. The first component always starts at zero. The range must be defined by some static integer value. However, the bounds need not be of the same integer type.

For example, given the following record type,

```
type Data_Record is
     record
         Data_Ready : Boolean;
         Data       : Integer range 0 .. 255;
     end record;
```

the following representation clause:

```
for Data_Record use
    record at mod 1;                        -- Single byte boundary
        Data_Ready at 0 range 0 .. 0;  -- First word, first bit
        Data       at 0 range 1 .. 15; -- First word, next 15 bits
    end record;
```

specifies that each object of this type starts on a single word boundary. The first bit of the word will contain the Data_Ready information, and the next 15 bits will contain the Data information. Note that 15 bits is more than enough space to hold the allowable integer values.

For each component of the record there can be one component clause. If none is specified the storage of that component is left to the implementation. The storage representation for components within variant records must not overlap. A component clause is only allowed for components when a constraint on the component is known at compile time.

One final note on representation clauses the entity must be declared in the same declarative part, package specification, or task specification as the representation clause for that entity.

Problem

In a communication network of a distributed system, the Asynchronous Communications Interface Adapter (ACIA) provides the formatting and control information necessary to interface serial asynchronous communications. The ACIA contains the status of the current state of communication buffers.

Specifically, in the ACIA:

* Bit 0 indicates the state of the Receiver Data Register, zero when the register is empty;
* Bit 1 indicates the state of the Transmitter Data Register, one when the register is empty;
* Bit 2 indicates the presence of the data carrier;
* Bit 3 indicates whether or not the input status of an interfacing modem is clear;
* Bit 4 signals a framing error, i.e., the absence of the stop bit in the message, resulting in a synchronization error, faulty transmission, or a Break condition;
* Bit 5 signals an overrun error when a character was received but not read prior to another character being received;
* Bit 6 signals parity error; and
* Bit 7 signals that an interrupt request has been received.

Some of this information is needed by the communications system which sends and receives messages. Assume that the ACIA is located at address 8#27# of the target machine and that the target machine has four bits per byte, and two bytes per word. Write the necessary code for representing the ACIA in Ada.

Discussion and Solution

First we need to represent the values of the bits in the ACIA, i.e., High and Low (1 and 0). We do this as follows:

```
type Status_Bit is (Low, High);
for Status_Bit use (Low => 0, High => 1);
```

Next, we define at an abstract level the structure of the ACIA. We can do this one of two ways. The first, as a record, follows:

```
type ACIA_Type is
    record
        Receiver_Data_Register    : Status_Bit;
        Transmitter_Data_Register : Status_Bit;
        Data_Carrier              : Status_Bit;
        Clear_To_Send             : Status_Bit;
        Framing_Error             : Status_Bit;
        Overrun_Error             : Status_Bit;
        Parity_Error              : Status_Bit;
        Interrupt_Request         : Status_Bit;
    end record;
```

The second, as an array, follows:

```
type Register_Bits is (Receiver_Data_Register,
                       Transmitter_Data_Register,
                       Data_Carrier,
                       Clear_To_Send,
                       Framing_Error,
                       Overrun_Error,
                       Parity_Error,
                       Interrupt_Request);

type ACIA_Type is array (Register_Bits) of Status_Bit;
```

The choice between these two representations to use in the system depends on how the ACIA is used within the program. One could argue that since each component of the ACIA is of the same type, an array is more appropriate. However, it may be more meaningful to access components by writing

```
ACIA_Register.Receiver_Data_Register
```

rather than by writing

```
ACIA_Register(Receiver_Data_Register)
```

in which case the record representation should be used. Another consideration might be whether or not the algorithm will loop through the register bits checking their values. If so, the array is the logical representation.

At this point the use of the ACIA is unknown, so for the purpose of our exercise, we will show the necessary representation clauses for both of these representations.

The record requires the following record representation clause to map each component of the record type to a single bit of the ACIA Register:

```
for ACIA_Type use
    record at mod 2;                  -- Double byte boundary
        Receiver_Data_Register    at 0 range 0 .. 0;
        Transmitter_Data_Register at 0 range 1 .. 1;
        Data_Carrier              at 0 range 2 .. 2;
        Clear_To_Send             at 0 range 3 .. 3;
        Framing_Error             at 0 range 4 .. 4;
        Overrun_Error             at 0 range 5 .. 5;
        Parity_Error              at 0 range 6 .. 6;
        Interrupt_Request         at 0 range 7 .. 7;
    end record;
```

Recall that a byte in this system is 4 bits, so the record must start on a double byte boundary. Also recall that two bytes form one word, so each component is "at 0," meaning, in the first word of the data object. Each component is formed by a range of bits, in this case one bit per component. Now all objects of type ACIA_Type will take exactly one word of storage, and that storage will start on a double byte boundary.

Recall that there is no specific representation clause for arrays. For the array structure the length clause could be used as follows:

```
for ACIA_Type'Size use 8;
```

This sets aside exactly 8 bits of memory for objects of this type. Note that this does not require that the 8 bits start on any byte boundary. However, because the address clause will be used to specify the starting address of the 8 bits, the alignment is unnecessary.

The address clause used to associate the ACIA Register (regardless of the specific type representation) to the physical register, appears as:

```
ACIA_Register : ACIA_Type;
for ACIA_Register use at 8#27#;
```

The complete code for the solution using a record type specification follows:

```
type Status_Bit is (Low, High);
for Status_Bit use (Low => 0, High => 1);

type ACIA_Type is
    record
        Receiver_Data_Register    : Status_Bit;
        Transmitter_Data_Register : Status_Bit;
        Data_Carrier              : Status_Bit;
        Clear_To_Send             : Status_Bit;
        Framing_Error             : Status_Bit;
        Overrun_Error             : Status_Bit;
        Parity_Error              : Status_Bit;
        Interrupt_Request         : Status_Bit;
    end record;

for ACIA_Type use
    record at mod 2;              -- Double byte boundary
        Receiver_Data_Register    at 0 range 0 .. 0;
        Transmitter_Data_Register at 0 range 1 .. 1;
        Data_Carrier              at 0 range 2 .. 2;
        Clear_To_Send             at 0 range 3 .. 3;
        Framing_Error             at 0 range 4 .. 4;
        Overrun_Error             at 0 range 5 .. 5;
        Parity_Error              at 0 range 6 .. 6;
        Interrupt_Request         at 0 range 7 .. 7;
    end record;

ACIA_Register : ACIA_Type;
for ACIA_Register use at 8#27#;
```

The complete code for the solution using an array type specification follows:

```
type Status_Bit is (Low, High);
for Status_Bit use (Low => 0, High => 1);
```

```
type Register_Bits is (Receiver_Data_Register,
                       Transmitter_Data_Register,
                       Data_Carrier,
                       Clear_To_Send,
                       Framing_Error,
                       Overrun_Error,
                       Parity_Frror,
                       Interrupt_Request);

type ACIA_Type is array (Register_Bits) of Status_Bit;

for ACIA_Type'Size use 8;

ACIA_Register : ACIA_Type;
for ACIA_Register use at 8#27#;
```

EXERCISE 7.2

OTHER IMPLEMENTATION-DEPENDENT FEATURES

Objective

This tutorial introduces several implementation-dependent features not yet addressed, specifically pragmas and unchecked programming.

Tutorial

Pragmas, like representation clauses, are instructions to the compiler. Rather than specifying internal configurations, a pragma gives permission to the compiler to perform certain steps that could be advantageous in execution. The key word to note here is "permission". They are not directives that the compilers must follow. This is what makes pragmas implementation-dependent.

There are two kinds of pragmas, predefined and implementation-defined. Predefined or language-defined pragmas are described in the Ada Language Reference Manual. Implementation-defined pragmas are specific to the particular implementation under consideration. Some pragmas take parameters. In general, there are restrictions on where a given pragma may appear if it is to be recognized and observed. The language defines 14 pragmas. They are:

Controlled,
Elaborate,
Inline,
Interface,
List,
Memory_Size,
Optimize,
Pack,
Page,
Priority,
Shared,
Storage_Unit,
Suppress, and
System_Name.

The Ada Language Reference Manual states that a compiler must recognize each of these pragmas, that is to say, it must not reject a program because it does not support the pragma which appears in the program. A warning should be issued and compilation continued.

This tutorial does not discuss all the predefined pragmas; rather, a few will be discussed to give the reader a feel for how pragmas could be used. For example, suppose a subprogram Search_Buffer is invoked repeatedly inside a loop, as in,

```
loop
    .
    .
    Search_Buffer (A);
    .
    .
end loop;
```

If execution speed is critical, a pragma could be used to eliminate the wasted object code used for calling the subprogram, returning from it, and passing parameters for each call, by specifying that the body of the subprogram replace the call. The predefined pragma Inline would be given in a statement in advance of the mentioned loop,

```
pragma Inline (Search_Buffer);
```

which requests that all calls to Search_Buffer be replaced by the subprogram body. The Inline pragma has to appear at a declaration point.

The Page pragma specifies that if the compiler is producing a compile listing, the text following the pragma should begin on a new page.

The pragma List takes one argument which is either On or Off. If On, a listing of the compilation will be printed until a List pragma with argument Off is encountered.

The pragma Optimize requests that the compiler perform optimization while producing the program's compiled code. It takes one argument, either Time or Space, to identify which to optimize.

Interface is used to allow Ada programs to call programs written in some other language, such as FORTRAN.

The pragma Suppress allows the compiler to omit certain runtime checks in the program which would otherwise be required. For instance,

Suppress (Range_Check)

allows the compiler to omit code which verifies that values assigned to certain objects satisfy the object's type constraints.

Implementation-defined pragmas are peculiar to a given implementation. These pragmas are not allowed to change the effect of the program, that is to say, the program should be compilable and achieve the same results with and without the pragma.

Another implementation-dependent feature in Ada is unchecked programming. There are two forms of unchecked programming, Unchecked_Conversion and Unchecked_Deallocation. Unchecked_Conversion is used to access the internal representation of data in order to change its interpretation. Unchecked_Deallocation is used to specifically control the deallocation of an object's storage.

In general, unchecked programming should be used sparingly because it makes the program less portable by tying the code to the specific target machine. However, if the program is concerned with device or operating system applications, a portable program is not a goal. Good engineering practice (and common sense) dictates that when unchecked programming is used in applications, its use should be localized. This minimizes the implementation-dependent areas of the system and increases the portability of the rest of the system.

Both Unchecked_Conversion and Unchecked_Deallocation are predefined generic library units in Ada. In order to use either of these features one must first import the library unit, via a with clause, and then instantiate it.

The typical Ada student may be wondering "Why does a language like Ada, which stresses strong typing, have a feature that allows the

programmer to perform unchecked type conversion?" The reason is that it is sometimes necessary in low-level programming to view a certain sequence of bits in different ways at different times, for instance as a sequence of flags at one point and an integer at another.

Ada's strong typing forbids type conversion between "unlike" types, for instance between an array (of bits) and an integer. Therefore, Unchecked_Conversion is used to convert a value of one type to a value of another. The form of Unchecked_Conversion is:

```
generic
    type Source is limited private;
    type Target is limited private;
function Unchecked_Conversion (S : Source) return Target;
```

Given the following:

```
type Bit_Type is (Off, On);
for Bit_Type use (Off => 0, On => 1);
for Bit_Type'Length use 1;
type Bit_Array is array (0 .. 3) of Bit_Type;
for Bit_Array'Length use 4;
Half_Byte : Bit_Array := (Off, On, On, On);
```

the instantiation of Unchecked_Conversion to convert the value stored in Half_Byte to an Integer value would appear as follows. A new integer type of the same length as Bit_Array has to be defined as well,

```
type Integer_4 is new Integer range 0 .. 15;
for Integer_4'Length use 4;
function Half_Byte_Conversion is new
        Unchecked_Conversion (Source => Bit_Array,
                              Target => Integer_4);
```

and a call to convert Half_Byte to an integer value would need to go through an intermediate step. The array is first converted to a value of type Integer_4 and then explictly converted to an integer.

```
Half_Byte_Value : Integer :=
            Integer (Half_Byte_Conversion (Half_Byte));
```

Note that one has to be very careful when using Unchecked_Conversion because it is just that, unchecked conversion. It is assumed that when it is used, it is used for a good reason and that the programmer is performing all the necessary checks.

The other form of unchecked programming, Unchecked_Deallocation, is used to deallocate the storage previously allocated for the storage of a variable. Note that allocation and deallocation refer to objects of an access type.

Normally, when objects are allocated,

```
type Acc is access Integer;
X : Acc := new Integer'(3);
```

storage for its value is set aside and associated with the access object (the access object contains the address of the storage). If there is not enough storage in the system for the allocated object, Storage_Frror is raised. Unchecked_Deallocation can be used to free up necessary space. The form of Unchecked_Deallocation is:

```
generic
    type Object is limited private;
    type Name is access Object;
procedure Unchecked_Deallocation (X : in out Name);
```

An instance of Unchecked_Deallocation using the type Acc declared above would appear as:

```
procedure Acc_Deallocation is new Unchecked_Deallocation
                (Object => Integer;
                 Name   => Acc);
```

and a call to deallocate the storage reserved for X's designated object would appear as:

```
Acc_Deallocation (X);
```

Note that Unchecked_Deallocation can be very dangerous. Once an object has been deallocated, it can no longer be used. After Unchecked_Deallocation, the deallocated access variable still exists, and still has a value, which is the address of the storage that was set aside for values. This storage however, may have been reused for other purposes, such as to hold other object's values. Often this storage will be reallocated to other programs in the operating system. Using the deallocated variable can have disastrous effects on the execution of all programs in the operating system, not just the program in which the object was deallocated.

When Unchecked_Deallocation is used, specific care must be taken to insure that the deallocated variable is no longer used in the system. Often this can be very involved when you have several access objects designating the same storage.

Utmost care must be used when Unchecked_Deallocation and Unchecked_Conversion are employed.

Problem

Answer the following questions.

1. The following package was developed for compiler A, which has an implementation defined pragma Private_Part. This pragma suppresses the listing of private parts of packages when given the argument Off.

```
pragma Private_Part (Off);
generic
    type Node_Type is (<>);
package Tree_Operations is

    type Tree_Type is limited private;
    procedure Add_Node (Tree : in out Tree_Type;
                        Node : in     Node_Type);

    procedure Delete_Node (Tree : in out Tree_Type;
                           Node : in     Node_Type);

    .
    .

private
    type Tree_Rec;
    type Tree_Type is access Tree_Rec;
    type Tree_Rec is
       record
           Data  : Node_Type;
           Left  : Tree_Type;
           Right : Tree_Type;
    end record;

end Tree_Operations;
```

Is this package illegal when compiled by another compiler that does not support the pragma Private_Part?

2. The designers of a particular program believe exceptions to have disastrous effects on the timing and efficiency of a critical system. They believe that their code performs all necessary checks and have included

```
pragma Suppress (Overflow_Check);
```

prior to some arithmetic computations in their program. During the testing of the program, when they attempted to verify that no exceptions were raised, Numeric_Error was raised. Is this a legal action for the compiler to take? If legal, why might the compiler have left in this check?

3. Given,

```
with Unchecked_Conversion;
procedure Convert_to_Lower_Case (Char : in out Character) is

    procedure Integer is new Unchecked_Conversion
            (Source => Standard.Character;
             Target => Standard.Integer);

    procedure Character is new Unchecked_Conversion
            (Source => Standard.Integer;
             Target => Standard.Character);

begin -- Convert_to_Lower_Case

    Char := Character (Integer (Char) + 32);

end Convert_to_Lower_Case;
```

What is the result when Convert_to_Lower_Case is called with the values ascii.bel, '+', and 'd'?

(The ASCII table is included here for your reference:)

```
type Character is

(nul,    soh,    stx,    etx,    eot,    enq,    ack,    bel,
 bs,     ht,     lf,     vt,     ff,     cr,     so,     si,
 dle,    dc1,    dc2,    dc3,    dc4,    nak,    syn,    etb,
 can,    em,     sub,    esc,    fs,     gs,     rs,     us,
 ' ',    '!',    '"',    '#',    '$',    '%',    '&',    ''',
 '(',    ')',    '*',    '+',    ',',    '-',    '.',    '/',
 '0',    '1',    '2',    '3',    '4',    '5',    '6',    '7',
 '8',    '9',    ':',    ';',    '<',    '=',    '>',    '?',
 '@',    'A',    'B',    'C',    'D',    'E',    'F',    'G',
 'H',    'I',    'J',    'K',    'L',    'M',    'N',    'O',
 'P',    'Q',    'R',    'S',    'T',    'U',    'V',    'W',
 'X',    'Y',    'Z',    '[',    '\',    ']',    '~',    '_',
 '`',    'a',    'b',    'c',    'd',    'e',    'f',    'g',
 'h',    'i',    'j',    'k',    'l',    'm',    'n',    'o',
 'p',    'q',    'r',    's',    't',    'u',    'v',    'w',
 'x',    'y',    'z',    '{',    '|',    '}',    '~',    del);
```

Discussion and Solution

1. No. A pragma which is not recognized by the compiler is ignored (with possibly a warning message). Also, implementation-defined pragmas can have no effect on the legality of a program: if the program is legal with the pragma, it must be legal without the pragma.

2. Yes. The Suppress pragma gives permission to the compiler to suppress certain checks, it does not require that the compiler suppress the checks. The compiler may continue performing the checks which could raise exceptions. For some machines, it is less efficient to suppress an overflow check than to allow it. For such machines, a compiler may well choose to ignore Suppress (Overflow_Check).

3. The following table defines the results of calls to Convert_to_Lower_Case.

Actual Value	Result
ascii.bel	'''
'+'	'K'
'd'	unknown

Note specifically this last result. The result of the computation Integer ('d') + 32 is outside the range of legal Ada characters. Ada does not define the effect of erroneous use of Unchecked_Conversion. i.e, when invalid values are converted to Character. A more appropriate conversion routine would be written in Ada as:

```
subtype Upper_Case_Character is Character range 'A' .. 'Z';

function Lower_Case_Character (Char : Upper_Case_Character)
                               return Character is
begin -- Lower_Case_Character

   return Character'Val (Character'Pos (Char) + 32);

end Lower_Case_Character;
```

INDEX

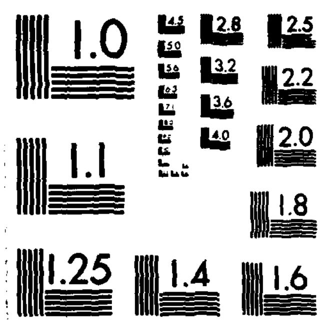
INDEX

ND
DATE
ILMED
-84
DTIC

AD-A146 257 ADVANCED ADA (TRADEMARK)(U) SOFTECH INC WALTHAM MA JUL 84 DAAB07-83-C-K514 5/5

UNCLASSIFIED F/G 9/2 NL

END
DATE
FILMED
12-84
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

SUPPLEMENTARY

INFORMATION

AD-A146257

DEPARTMENT OF THE ARMY
HEADQUARTERS US ARMY COMMUNICATIONS-ELECTRONICS COMMAND
AND FORT MONMOUTH
FORT MONMOUTH, NEW JERSEY 07703

REPLY TO
ATTENTION OF:

15 OCT 1984

Center for Tactical Computer Systems

Ms. Madeline Crumbacker
Defense Tactical Information Center
Cameron Station
Alexandria, Virginia 22314

Dear Ms. Crumbacker:

As per phone conversation with Ms. Andrea Cappellini, CENTACS on 11 October 1984, a copyright statement has been omitted on documents sent to DTIC and NTIS. Enclosed please find the copyright statement (Encl 1) that must appear in the enclosed list of document (Encl 2). If you have any questions, please contact Ms. Cappellini at 201-544-4280.

Sincerely,

Dennis J. Turner
for JAMES E. SCHELL
Director, CENTACS